CI/CD CORPORATE AUTOMATION PIPELINE USING DEVOPS IN AWS

A PROJECT REPORT

Submitted by

JUJJAVARAPU SUJAN CHOWDARY (RA2412033010001)
NAGA VENKATA SIVANIKHIL MARADANI (RA2412033010020)
NAVEEN KUMAR REDDY (RA2412033010029)

Under the Guidance of

Annapurani Panaiyappan.K
(Professor, Department of Networking and Communications)

In Partial fulfillment of the requirements for the degree of

MASTER OF TECHNOLOGY

CLOUD COMPUTING

DEPARTMENT OF NETWORKING AND COMMUNICATIONS
COLLEGE OF ENGINEERING AND TECHNOLOGY
SRM INSTITUTE OF SCIENCE AND TECHNOLOGY
KATTANKULATHUR - 603 203
DEC 2024

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

SRM INSTITUTE SCIENCE AND TECHNOLOGY

KATTANKULATHUR – 603 203

CERTIFICATE

This is to certify that the research report entitled "**CI/CD Corporate Automation Pipeline Using DevOps in AWS**" has been carried out by **Jujjavarapu Sujan Chowdary, Naga Venkata Sivanikhil Maradani, and Naveen Kumar Reddy** under my supervision and guidance.

This work is submitted in partial fulfilment of the department for the **Master of Technology in Cloud Computing, Department of Networking and Communication, at SRM Institute of Science and Technology, Kattankulathur,** during the academic year 2024.

The research embodies the candidate's original work, except where due references are made, and has not been previously submitted for the award of any degree or diploma

9/12/24

Dr. Annapurani Panaiyappan . K

(Supervisor)

Dept. of Networking and Communications Kattankulathur 603 203 SRM IST

M. Lakshmi

Dr. M. Lakshmi

(Head of the Department)

Dr.M.LAKSHMI,B.E.,M.E.,Ph.D.,
Professor & Head
Dept. of Networking and Communications
School of Computing
College of Engineering and Technology
SRM Institute of Science and Technology
SRM Nagar, Kattankulathur - 603 203,
Chengalpattu Dist, Tamil Nadu, India.

ii

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

DEPARTMENT OF NETWORKING AND COMMUNICATIONS

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

OWN WORK DECLARATION FORM

DEGREE COURSE: M-TECH - CLOUD COMPUTING

STUDENT NAMES: JUJJAVARAPU SUJAN CHOWDARY, NAGA VENKATA SIVANIKHIL MARADANI, NAVEEN KUMAR REDDY

REGISTRATION NO.: RA2412033010001, RA2412033010020, RA2412033010027

TITLE OF WORK: - CI/CD CORPORATE AUTOMATION PIPELINE USING DEVOPS IN AWS

We hereby certify that this assessment complies with the University's Rules and Regulations relating to Academic misconduct and plagiarism, as listed in the University Website, Regulations, and the Education Committee guidelines. We confirm that all the work contained in this assessment is my / our own except where indicated, and that We have met the following conditions:

- Clearly referenced / listed all sources as appropriate
- Referenced and put in inverted commas all quoted text (from books, web, etc)
- Given the sources of all pictures, data etc. that are not my own
- Not made any use of the report(s) or essay(s) of any other student(s) either past or present
- Acknowledged in appropriate places any help that We have received from others (e.g. fellow students, technicians, statisticians, external sources)
- Compiled with any other plagiarism criteria specified in the Course handbook / University website

We understand that any false claim for this work will be penalized in accordance with the university policies and regulations.

DECLARATION:
We are aware of and understand the University's policy on Academic misconduct and plagiarism and We certify that this assessment is our own work, except where indicated by referring, and that we have followed the good academic practices noted above.
If you are working in a group, please write your registration numbers and sign the date for every student in your group.

J. Sujan Chowdary. M. Nikhil Naveen kumar

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR – 603 203

ACKNOWLEDGEMENT

We take opportunity to express our heartfelt gratitude to everyone who contribute to the successful completion of this research paper, as no research work is ever a solitary endeavour. It is the outcomes of collaboratives efforts, guidance, and support.

We extend our deepest gratitude to our **Research Supervisor** (**Dr. Annapurani Panaiyappan .K**), for their invaluable guidance, encouragement, and throughout this research. Their expertise and insights have been instrumental in shaping this work and providing clarity at every stage.

We would like to thank the **Department of Networking and Communications**, **SRM Institute of Science and Technology, Kattankulatur**, for providing us with the necessary resources and an environment conducive to research.

ABSTRACT

It is the design and implementation of a Continuous Implementation and Continuous Deployment (CI/CD) automation pipeline based on DevOps practices within Amazon Web Services (AWS). The primary objective of this work is to design and implement principles of a strong, secure, and efficient CI/CD pipeline fitting enterprise applications, realizing smooth code integration, running automated tests, vulnerability assessments, secure deployment, and ultimately efficient monitoring. Therefore, adopting scalable cloud-based CI/CD is very fundamental to the new wave of continuous integration and delivery in modern software development to keep up with qualitative code quality and rapid deployment. This report will explain step by step development of CI/CD pipeline by including tools and technologies such as Jenkins, Docker, Kubernetes, SonarQube, Nexus, and monitoring systems like Prometheus and Grafana. This offers a unified approach in the direction of streamlining and automating the software delivery process.

The design of the setup will include setting up secure infrastructure in AWS, configuring the necessary networking ports, and deploying virtual machines to host key services. Using Jenkins, the pipeline setup initiates a sequence of operations set, comprising code compilation, Trivy for vulnerability scanning, and quality analysis for code with SonarQube. The applications are kept isolated through docker containers to enable flexible management and efficient resource usage. Then, Kubernetes maintains orchestration of the containers across nodes to achieve secure, reliable deployment and scaling of applications. Security is addressed at several levels of the tool, from role-based access control to vulnerability scanning, with automated alerts and notifications to address potential threats. Prometheus and Grafana are implemented to get real-time monitoring of both the health of the application and system metrics so that issues can be addressed and performance optimized in advance.

Monitoring solution integration provides systemic health insights that help continuously improve the resilience and uptime of the systems. This is the demonstration of the scalability aspect of an enterprise DevOps team pipeline, which depicts agility, collaboration, and operational efficiency in all the workflows offered through software development. The outputs thus show how automating CI/CD in a cloud environment optimizes resource utilization and creates a basis for scalable, secure, and efficient delivery of software. This study has great practical value for the field of DevOps, given its detailed blueprint for adoption in CI/CD automation by organizations in cloud environments. The integration of security checkpoints, automated testing, and continuous monitoring leads to high performance architecture of CI/CD pipelines. Future work might extend this architecture to adapt it with AI-driven optimizations for adaptive CI/CD workflows.

TABLE OF CONTENTS

Page.No

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

In today's fast-evolving software development organizations must focus on deploying, monitoring, and securing applications efficiently if they want to remain competitive. This requires the practice of CI/CD as part of the DevOps methodology for continuous evolution. Modern software delivery is all about the CI/CD pipeline because it forms the backbone in totally automated stages of building and testing an application toward eventual deployment for users to enjoy newly developed features, updates, or security patches quickly and reliably. This project encompasses a solution on the design and implementation of a corporate CI/CD pipeline in cloud deployment, utilizing DevOps in an Amazon Web Services environment while tackling difficult industry-specific problems involving automation, security, and monitoring.

As organizations increasingly move to cloud environments, they need robust, automated workflows that minimize the time and effort required for software deployment. With cloud providers such as AWS offering an array of services tailored for CI/CD, many companies now have the opportunity to leverage these tools to improve software quality and operational efficiency. This comes along with the challenges of embracing CI/CD and DevOps in cloud environments in the presence of secure infrastructures set up, effective monitoring configurations, and complete security check implementations. Recent studies indicate that most organizations are still unable to tap into the full potential of CI/CD and DevOps, mainly because of minimal expertise, configuration complexity, and lack of optimized security measures undertaken to address these challenges by providing a guide on creating a pipeline tailored for corporate environments on AWS. Included in the report, therefore, are the explorations of security measures, testing frameworks, and monitoring setups critical to production but commonly missing in the current literature. This pipeline implementation sets out to demonstrate best practices to deploy with streamlined processes while maintaining high standards of security and compliance.

The central question driving this project is: How can one implement a secure, robust, and scalable CI/CD pipeline in the corporate environment by using AWS and DevOps practices in order to boost the efficiency of deployment and enhance resilience in operation? This question attracts our double focus on technical implementation and also security within the pipeline of CI/CD.

This calls for design and development of the CI/CD pipeline leveraging DevOps approaches on the AWS cloud platform to smoothly deploy in corporate applications. Inculcate testing and security checkpoints: Test and scan automatically for vulnerabilities to ensure the highest quality and security standards of every code deployed. Implement a Secure Infrastructure to create a secure AWS infrastructure that can support the pipeline. Network configuration and access controls and role-based permissions are examples. Automation of Notification and Monitoring Systems to configure real-time monitoring and alert systems to observe the health of applications as well as infrastructure performance. This objective also involves putting in place an extended email notification system to keep stakeholders aware of deployment statuses and issues.

With pipeline configuration for Docker and Kubernetes containers and usage of Kubernetes clusters inside the CI/CD, it's possible to maximize deployment flexibility and extend deployment scalability, especially those concerning microservices-based applications.
Industry-standard DevOps tools; Jenkins, SonarQube, Nexus, and Prometheus, to make pipelines perform at their best to provide continuous monitoring and reliable application-delivery capabilities.
The goals were designed with utmost care to ensure a thorough understanding of CI/CD pipeline automation while taking into consideration crucial security, efficiency, and scalability concerns.

This report will attempt to apply the CI/CD pipeline within an AWS environment by employing industry-standard DevOps practices. Although other cloud providers offer similar services, this report selected AWS because it is mature, scalable, and used across various industries. Also, though the CI/CD pipeline will employ Docker, Kubernetes, and Jenkins, the project will not delve into other alternatives for tools or infrastructure except these three main tools. Its scope will be restricted to AWS and these tools, an easily replicable model that other organizations can take up with minimal configuration.

In addition, the project further focuses its attention on the secure practice in deploying and monitoring, deliberately leaving out all areas such as code refactoring and methodologies applied in software development. With the avoidance of these areas of work, the project can be assured to remain concentrated on its main goal of automating the infrastructure security and monitoring setup.

In both academia and industry, practical application and impact would form the basis of this study. The reason for increasing importance in CI/CD pipeline automation is that it relates more and more to business needs, especially as its size of software deployment operations needs to grow to accommodate more global users. In the pursuit of filling the gaps that may exist in existing literature, we propose this report that details a real-world approach to building and deploying a secure CI/CD pipeline on AWS. It was recently identified as one of the recurring challenges in many recent studies.

1.1 Objective

This report is important in the industry for IT professionals looking for efficient ways to manage and secure their deployment processes. Our findings and methodologies are intended to help decision-makers make a more informed choice about adopting DevOps practices with a clearer understanding of the technical and security considerations involved. The CI/CD pipeline model presented here could, therefore, be the very foundation architecture that organisations take and adapt and grow depending on their unique needs thus making this project of very much value to promoting innovation in corporate DevOps strategy.

The project "CICD Corporate Automation Pipeline Using DevOps in AWS" aims to design, implement, and optimize a comprehensive CI/CD pipeline. More specifically, the overall objective is to streamline the corporate software delivery lifecycle through automation, especially using the capabilities of DevOps practices and AWS cloud infrastructure. This project intends to reduce the intervention rate of humans, improve software quality, enhance the speed of deployment, and

maintain the highest security and reliability standards at each phase of development and deployment.

1.2 Problem Statement

Critical need drivers: Software delivery mechanisms have become even faster and more efficient as part of the technology landscape for rapid change. In addition, traditional software delivery pipelines will remain bottlenecked, delayed, and inconsistent with regard to manual processes. By automatically managing all these processes through CI/CD practices, the project expects the teams to be more able to focus on inventions and development instead on boring repetitive tasks. This is consistent with trends and best practices as outlined by studies such as those published by Rubert et al. [5] and Wiedemann et al. [16], who point to the tangible advantages of implementing DevOps in terms of improving quality of software and productivity of the team.

We focus on integrating robust testing and security checkpoints at various stages of the pipeline. In the modern corporate landscape, software applications face constant threats from vulnerabilities and cyberattacks. By incorporating tools such as Trivy for vulnerability scanning and SonarQube for code quality analysis, the pipeline ensures that every deployment meets stringent security standards. That is to say, the adoption of such security measures demonstrates the proactivity towards risk mitigation that Wong et al. [6] propose advocating for containerized security practices in software development.

1.3 Project Tools

The efficient configuration and utilization of tools like Jenkins, Docker, and Kubernetes. These are some of the critical enabling technologies for a seamless CI/CD process. Master-slave nodes setup, application deployments with Kubernetes clusters, and managing containers with Docker are some of the pivotal tasks included in designing the pipeline. Such configurations ensure scalability, fault tolerance, and easy management of the distributed systems.

1.4 Monitoring

It creates solid monitoring and notification systems utilizing Prometheus, Grafana, and enhanced email configurations. Therefore, the health and performance of applications and infrastructure would continuously be tracked, allowing for an effective response to any abnormal behavior. The use of integration monitors therefore shows a commitment towards delivering excellence in operations, so this pipeline is not only installed correctly but also sustained through continuous performance and reliability. The aim of the project is to thereby provide a model framework of CI/CD pipelines that can be replicated or adapted in varying corporate environments.

CHAPTER 2

LITERATURE REVIEW

This has, in turn, fueled an interest in DevOps, CI/CD pipelines, and deployment frameworks for cloud-based systems. This review assesses the current theories, frameworks, and models in CI/CD and DevOps research, analyzing relevant studies which have contributed to best practices in practice and identifying areas that remain unexplored.

This helped conceive CI/CD, as a cure for age-old software development methodologies. That rarely could adapt or sustain the demands and tempo of modern applications [1]. DevOps changed all the scenario where today's integration has managed to enhance collaboration, streamline workflow, and deploy application in an integrated process[2]. According to Grande et al, the productivity and quality of code have significantly improved with the introduction of DevOps practices in global software engineering. However, depending on the specific tools and practices adopted, these benefits may vary [1]. Docker-based containerization and Kubernetes orchestration have been the basic building blocks in the modern CI/CD pipelines. It enables Docker, which allows for isolated environments for application deployments, while Kubernetes offers scalable and reliable container orchestration [3]. Shan et al's KubeAdaptor framework presented how the Kubernetes can be leveraged in workflow containerization, thereby making it uniquely suitable for CI/CD pipelines. This model brought about new capabilities in big deployments; however, to achieve the same, certain complex configurations are needed which are deterrents to wider adoption.

Many studies focus on the security concerns related to DevOps and CI/CD. For instance, Wong et al. published a comprehensive threat analysis related to containerized environments; in this work, threat models and mitigation strategies are proposed [6]. Such an article highlighted the role of vulnerability scanning and access controls within CI/CD pipelines. According to Doan and Jung, however, existing vulnerability scanning tools like Trivy would need further maturity to recognize even complex multi-stage vulnerabilities as an area of gap in overall vulnerability assessment [2].

Several models appeared in the literature, including frameworks and best practices to implement CI/CD in a corporate environment. Rubert and Farias conducted a case study on how continuous delivery impacts code quality, finding automation within the CI/CD process to drastically reduce defect rates and software quality improvement [5]. Their work falls in the findings done by Lange et al. They discussed automation's place in CI/CD pipelines and further suggested tools that enable easy deployments for enterprise systems [4] as shown in table 2.1.

Table 2.1. Summary of Relevant Models and Experiments.

Focus Area	Limitations
DevOps in Global Engineering	Challenges in standardizing tools
KubeAdaptor for Workflow Containerization	Requires advanced Kubernetes expertise
Security in Containerized Environments	Limited scope on multi-stage vulnerabilities
CI/CD and Code Quality	Limited focus on infrastructure security
Vulnerability Scanning	Needs enhanced capabilities for complex vulnerability stages
CI/CD Tool Selection	Limited to tool-based perspectives

Another recurring theme, however, was that of robust security within the CI/CD pipeline. Mills et al., conducted one of the longest studies; they grounded their case for continuous rather than all-at-once risk-based assessments of Docker container vulnerabilities further lowering the chances of risks due to software-vulnerability related issues in production [9]. Their effort aside, most of these studies focus on point-in-time initial security, leaving little to actual continuous periodic security evaluations. In such a scenario where containerization is growing rapidly, what is needed is a dynamic, continuous security framework, an area addressed by this research with the introduction of security checkpoints within the deployment lifecycle.

Besides this, Zhao et al. elaborated on the time-sensitive applications of the cloud and highlighted the requirement for reliable and automated deployment pipelines to eliminate latency as a time-dependent application requires [7]. In line with this, this paper integrates the automated testing and scanning by security throughout the pipeline to comply and make any risk proactive. Table 1. explains the limitations of existing pipelines and problem statements are framed from it.

In an efficient CI/CD pipeline, there is a need for strong tools and monitoring systems that track the deployment process and application health. According to Lange et al. in their research about build automation, Jenkins is considered one of the most significant tools applied in CI/CD due to its powerful plugin ecosystem, whereby services like Docker and Nexus can quickly be customized and integrated [4]. Devops control approach by Wiedemann et al. is considered to be based on continuous monitoring; therefore, continuous monitoring improves the visibility of applications deployed into operations.

Both Prometheus and Grafana are widely suggested in literature for real-time monitoring. With the two, an estimate of the system's metrics as well as application performance could be deduced. The two tools are critical to deciding on bottlenecks as well as whether to optimize resource utilization. Often, literature does not give one all-inclusive guidelines about configuring such monitoring systems in a corporate CI/CD pipeline. This gap is covered by the current research in terms of an integrated monitoring solution.

While the theoretical and experimental studies are unveiling important aspects of CI/CD and DevOps practices, there are still various gaps. For instance, there is very limited literature on the

end-to-end setup of CI/CD pipelines, especially within AWS environments. This is because most existing studies focus on isolated tools and practices rather than a cohesive framework [13]. Among the highly discussed topics, only a few studies are found about continuous vulnerability scanning and access control within the CI/CD lifecycle, especially when Docker and Kubernetes are used for containerization and orchestration [9].

Moreover, various research papers discuss DevOps practices for small to medium-sized organizations or isolated applications but the intricacies of a corporate environment demand something stronger and more scalable solutions [5]. Finally, despite the relevance of monitoring as discussed in the related studies, hardly any literature can be seen that delivers actual steps in integrating these tools into the CI/CD pipeline besides other DevOps tools.

CHAPTER - 3

DESIGN OF CI/CD AUTOMATION PIPELINE

This section details the methodology employed to build a corporate CI/CD pipeline in AWS. The outline of project approach, tools, configurations, and processes had been undertaken in building a CI/CD pipeline. Intensive testing with rigorous secure infrastructure setup and various security checks across the entire CI/CD process were followed in the design of this pipeline in order to ensure it was secure, automated in a corporate environment, scalable, and efficient with real-time monitoring.

3.1 Project Design

The applied project approach was employed to steer the study towards some practical challenges that could and must be exercised in the implementation of DevOps within the corporate CI/CD pipeline. This enables a systematic iterative development and optimization of the pipeline, giving it room to check different configurations and tools towards achieving the most efficient and secure setup.

3.1.1 Project Approach

A mixed-methods approach was used, combining qualitative assessments with quantitative data analysis. This enabled the evaluation of both the functional and security aspects of the pipeline while allowing for flexible problem-solving to refine the pipeline based on real-world needs. The qualitative aspect involved examining CI/CD frameworks, while quantitative methods focused on performance, security metrics, and deployment speed.

3.1.2 Data Collection

Data was collected during each stage of the pipeline setup and testing process, including configuration times, error rates, security vulnerabilities identified, and deployment success rates. This data helped determine the effectiveness of each tool and configuration choice. Monitoring tools like Prometheus and Grafana provided additional metrics on system performance and helped fine-tune the pipeline setup.

3.2. Infrastructure Setup and Tool Selection

It was important to ensure the right tools and to establish a secure, scalable infrastructure for the report. Every component in the pipeline was chosen for functionality, ease of integration, and security features.

3.2.1 AWS Infrastructure Setup

The pipeline was deployed on AWS, leveraging its scalability, security, and ease of integration with DevOps tools. AWS EC2 instances were created for each component of the pipeline, including tools. For security, the virtual private cloud (VPC) setup in AWS was configured to limit access, using private subnets where possible, and deploying security groups to restrict port access based on the needs of each service as shown in table 3.1.

Table 3.1. AWS components used and their specifications

AWS Component	Specification
EC2 Instances	t3.medium (2 vCPU, 4 GB RAM)
Elastic Load Balancer (ELB)	Application Load Balancer
S3 Bucket	Standard storage class
VPC (Virtual Private Cloud)	Custom VPC with private and public subnets
IAM Roles and Policies	Custom IAM roles for Jenkins, EC2, S3, etc.
RDS (Relational Database Service)	db.t3.micro (1 vCPU, 1 GB RAM)
CloudWatch	CloudWatch Logs and Alarms
Elastic File System (EFS)	Standard Storage
Route 53	Hosted Zone
Elastic Container Registry (ECR)	Standard ECR registry
Secrets Manager	Standard Secrets Manager
SNS (Simple Notification Service)	Standard SNS topic

3.2.2 Tool Selection and Configuration

To meet the report's objectives, specific tools were chosen based on their functionality in CI/CD pipelines:

Jenkins was selected for its flexibility in managing CI/CD workflows, plugins, and integration with other tools like Docker and Kubernetes.

Jenkins is one of the most widely used automation servers for the continuous integration and continuous delivery process in DevOps environments. Jenkins can be exploited by experts in a wide range of workflows and technology stacks, mainly because it has a huge plugin ecosystem that supports plugins for most development, testing, and deployment tools.

Jenkins helps in automation of the CI/CD cycle ranging from the code integration phase through to testing and deployment as shown in Fig 3.1. Jenkins auto-triggers builds based on changed entries in the repository because of direct integrations with version control systems, making sure systematically that a build will hold the latest code that would be tested and deployed. Jenkins extensibility through plugins ensures that it can harmoniously integrate with all the other core CI/CD tools, ensuring that it continues to be an adaptable core component in DevOps pipelines in Table 3.2.

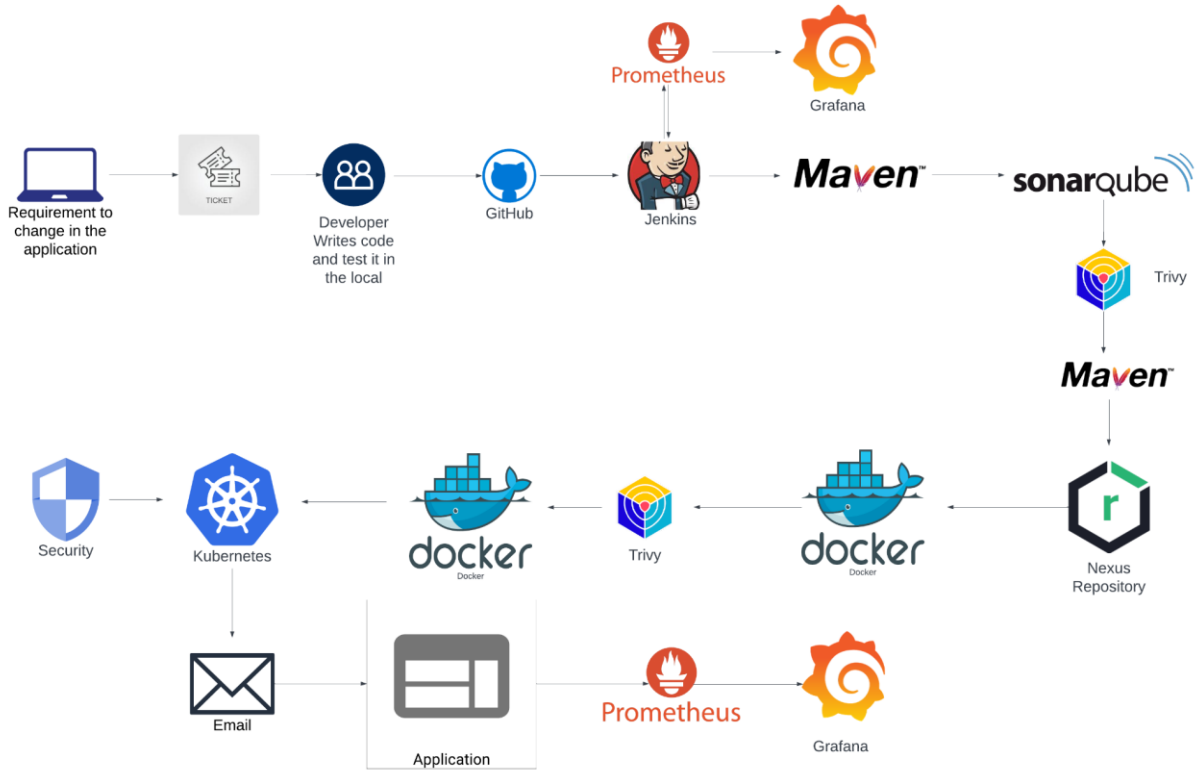

Fig. 3.1. CI/CD pipeline Architecture

Table 3.2. Jenkins CI/CD Pipeline Stages

Stage	Purpose
Git Checkout	Retrieves the report code from the source control repository (Git).
Compile	Compiles the source code to check for syntax errors and ensure code correctness.
Test	Runs unit and integration tests to verify functionality and identify issues early.
File System Scan	Scans the report filesystem for vulnerabilities, ensuring security compliance.
SonarQube Analysis	Analyzes code quality, security, and adherence to coding standards using SonarQube.
Quality Gate	Ensures code meets pre-defined quality standards before proceeding to the next stage.
Build	Packages the compiled code into an artifact (e.g., JAR/WAR) for deployment.
Publish to Nexus	Uploads the packaged artifact to the Nexus repository for version control and sharing.
Build & Tag Docker Image	Creates and tags a Docker image for application deployment.
Docker Image Scan	Scans the Docker image for vulnerabilities, ensuring secure deployment.
Push Docker Image	Uploads the Docker image to a container registry, making it available for deployment.
Deploy to Kubernetes	Deploy the application Docker image to the Kubernetes cluster.
Verify Deployment	Checks that the deployment succeeded by verifying the status of Kubernetes pods/services.
Notification	Send an email notification with the build status and details.

Docker was used to create containers, allowing for isolated environments for each microservice and easy deployment across various environments.

This technology known as Docker containerizer all an application's dependencies and applications inside a container, hence offering consistency across different environments. Containers are lightweight, portable, and reproducible; something critical for applications that demand stability in all three aspects: development, staging, and production.

By encapsulating the environment-specific configurations and dependencies inside containers, Docker enables consistent deployment of applications in CI/CD workflows. This allows builds to be executed within isolated environments whereby issues caused by dependency conflicts can be minimized as well as the process of deploying streamlined. Docker can even integrate with tools like Kubernetes and manage containers in a clustered deployment for efficient doing so.

Kubernetes provided container orchestration and enabled the scalability needed in corporate environments.
An open-source platform for running distributed clusters of containerized applications, Kubernetes automates the deployment, scaling and management of containerized applications, providing features like load balancing, self-healing, and service discovery.

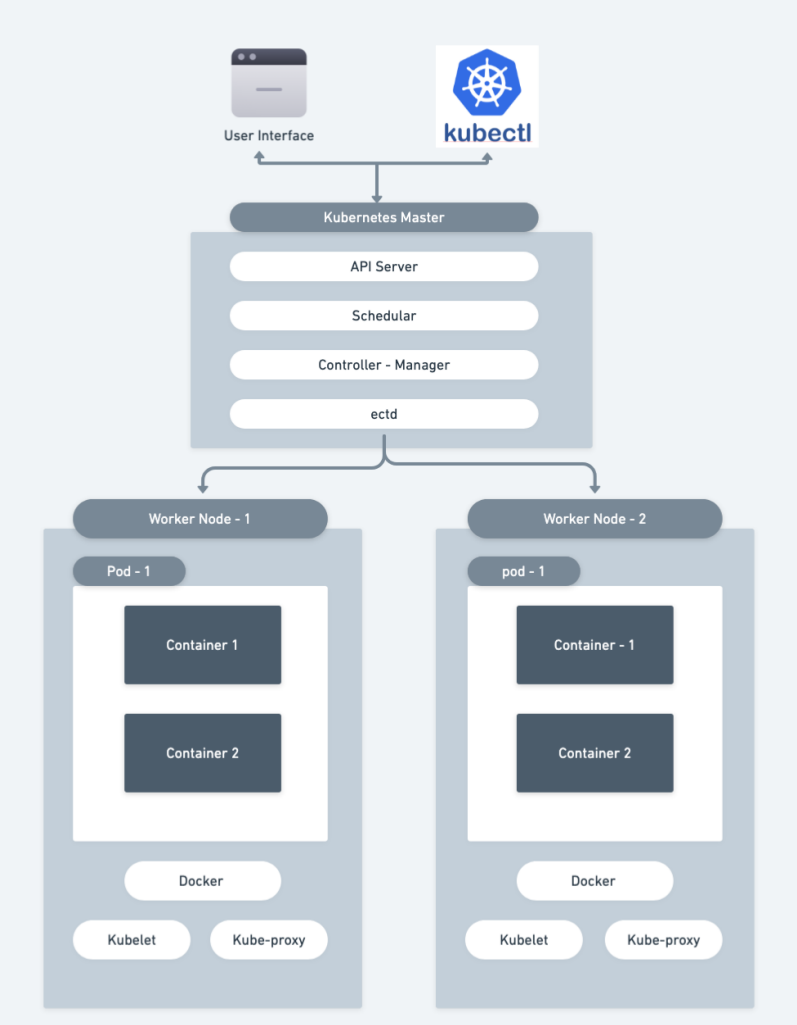

Fig. 3.2. Kubernetes Node's Architecture

Based on CI/CD, Kubernetes is used to manage the deployment of scalable containerized applications as shown in Fig 3.2. Because the applications are managed within a cluster, it ensures resilience and load distribution for those applications. It maintains the availability and stability of an application through automated rollouts and rollbacks, which are quite critical in production-ready CI/CD pipelines.

SonarQube was chosen for code quality analysis, as it offers insights into potential code issues, bugs, and vulnerabilities. SonarQube is a continuous source code quality inspector that detects bugs and vulnerabilities in addition to issues of maintainability using source code; it supports various programming languages and reporting mechanisms.

SonarQube integrates with CI/CD pipelines so that code quality analysis forms an automated part of the build. The checks are done by running SonarQube against defined quality criteria. When this is done, SonarQube maintains quality standards of the code within different development cycles. This means possible issues will be flagged prior to deployment, which strengthens code reliability and maintainability.

Nexus served as a repository manager, enabling artifact management and secure access control.
Nexus is a repository manager to store, retrieve, and manage any form of artifacts to build or deploy the software system. Further, it can be considered as the central place for managing binaries and dependencies that are generally used to compute build artifacts.

Nexus provides artifact management within CI/CD pipelines, wherein it helps store and retrieve build artifacts throughout the development lifecycle. For example, a Docker image or a Java library could be considered build artifacts. It helps centralize dependencies, thus providing effective build processes and version control over the artifacts in question to ensure consistency across deployments.

Prometheus and Grafana were used for monitoring and visualization of metrics, essential for real-time pipeline management.
Prometheus is an open-source monitoring and alerting toolkit that is scalable and reliable, whereas Grafana is a visualization platform which is used in conjunction with Prometheus to give real-time insights into how everything is performing in your application and infrastructure.
Thus, Prometheus and Grafana enable monitoring and visualization of infrastructure together. Prometheus collects metrics from differing services within the CI/CD environment and visualizes them through customization dashboards using Grafana. The monitoring layer provides the crucial feedback for system health, load, and performance. This makes maintenance proactive, as well as increases response times to any potential issues in various components and systems of the infrastructure.

AWS offers a full range of cloud services to support CI/CD infrastructural requirements on computing, storage, networking, and monitoring. Mostly, AWS components like EC2, S3, ELB, VPC, and CloudWatch are used in CI/CD configuration.

The CI/CD pipeline infrastructure is supported with trustworthy, scalable computing, storage, and monitoring solutions by AWS infrastructure. Critical components such as Jenkins, Docker, and SonarQube are hosted on EC2 instances, while build artifacts and backups are kept in S3. With CloudWatch for monitoring and logging, pipeline performance and resource utilization visibility is critical in maintaining efficiency and stability within cloud-based CI/CD workflows.

3.3. CI/CD Pipeline Implementation Process

Implementing the pipeline of CI/CD required setting up numerous stages that branched out from initial configurations all the way to deploying a secure, automated build and deployment process.

3.3.1 Jenkins Setup and Configuration

Jenkins was set as the central CI/CD tool. After installing Jenkins on an EC2 instance, crucial plugins for Docker, Kubernetes, SonarQube, and Nexus integration were installed. Master and slave nodes were configured to ensure that distributed build processes can be carried out. This is quite necessary for large builds management.

3.3.1.1 Pipeline Creation

Pipeline structure along with version-control pipeline was available using Jenkins pipelines with declarative syntax as shown in table 3.3. The pipeline was designed to have multiple stages involved, such as compilation of code, testing, scanning for security, artifact storage, and deployment as shown in Fig 3.3. These stages were configured to handle failures at every checkpoint so that feedback may occur towards immediate corrections and improvements in the code.

Table 3.3. CI/CD Pipeline Stages and Actions

Stage	Actions/Tools Used
Git Checkout	git plugin, Git credentials
Compile	Maven compile
Test	Maven test
File Scan	Trivy fs
SQ Analysis	SonarQube, sonar-scanner
Quality check	SonarQube Quality Gate
Build	Maven package
Publishing into Nexus repo	Nexus, Maven deploy
Build & Tag Docker Image	Docker, docker build
Docker Scanning Image	Trivy image
Push	Docker, docker push
Deploying in Kubernetes	Kubernetes, kubectl apply
Verification	Kubernetes, kubectl get pods/svc
Notification	Jenkins email-ext, HTML email template

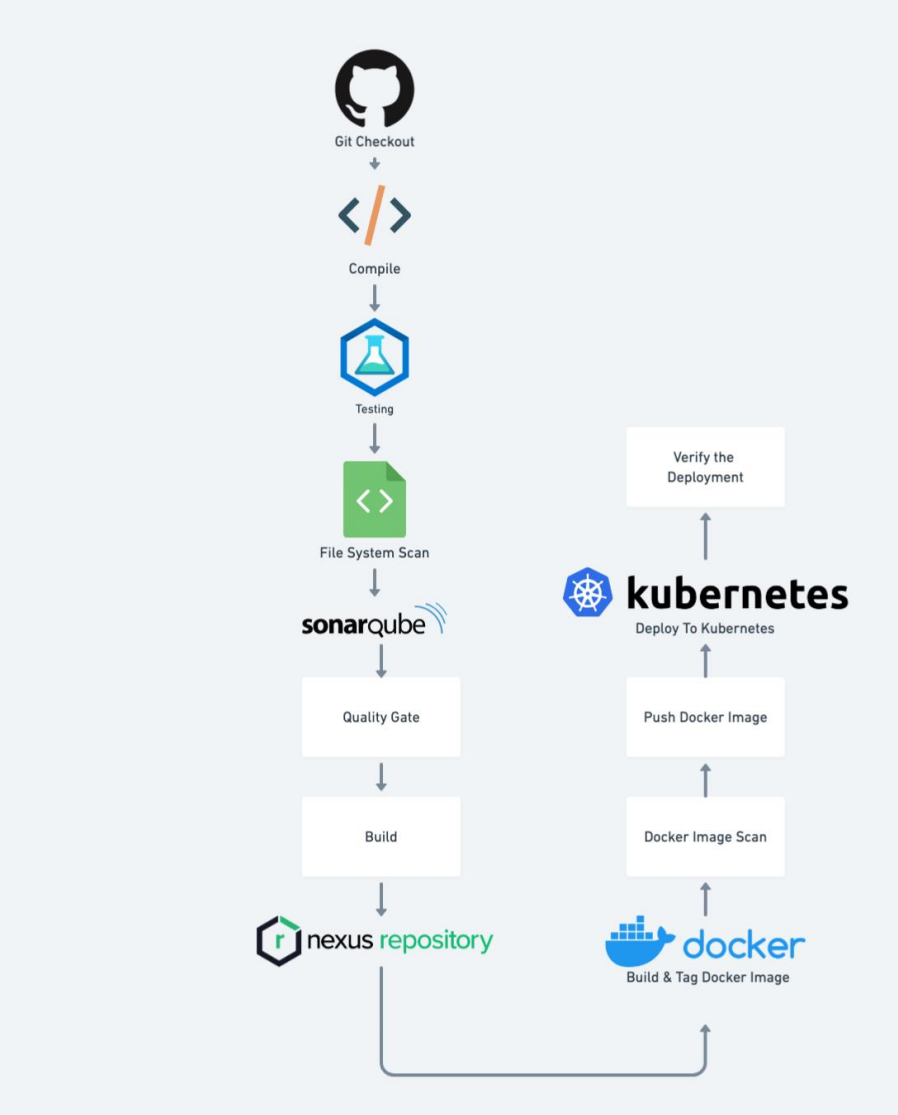

Fig. 3.3. Pipeline Stages

3.3.2 Docker and Kubernetes Setup

The Docker platform was installed on each of the EC2 instances for creating isolated, manageable environments for each component, like Jenkins, SonarQube, and Nexus. Using Docker, containers were created for encapsulation of those services to be able to scale and manage them very easily.

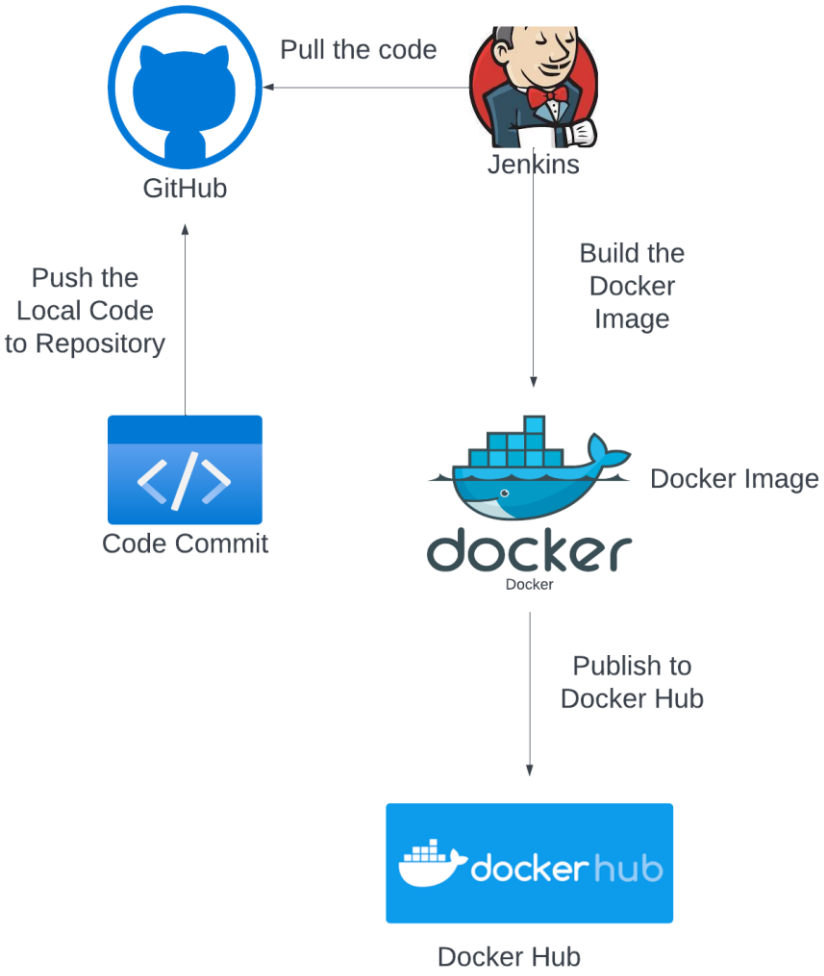

Fig. 3.4. Jenkins CI/CD with Github integrated on AWS EC2 Instance

Kubernetes was deployed to orchestrate Docker containers as shown in Fig 3.4. The cluster was initialized and configured using `kubeadm`, and each service in the Kubernetes cluster was assigned pods, which led to efficient allocation of resources and load balancing. Kubernetes was scaled to high-traffic corporate environments with the required scalability.

3.3.2.1 Security Configurations

This was implemented in Kubernetes through role-based access control, which ensures users and applications have only necessary permissions. Pod security policies were also implemented to ensure that a compromised container will not lead to any malicious outcome. Instance In-Bound security groups as shown in Table 3.4.

Table 3.4: In-Bound Security Groups

TYPE	PROTOCOL	RANGE	SOURCE
SMTP	TCP	25	IPV4
Custom TCP	TCP	3000-10000	IPV4
HTTP	TCP	80	IPV4
HTTPS	TCP	443	IPV4
SSH	TCP	22	IPV4
Custom TCP	TCP	6443	IPV4
SMTPS	TCP	465	IPV4
Custom TCP	TCP	30000-32767	IPV4

3.4. Security Measures and Testing

Security was one of the highlighted focus points while implementing the pipeline. There are lots of security mechanisms integrated through the pipeline at various levels to ensure that the integrity of the pipeline and applications being deployed is maintained.

3.4.1 Vulnerability Scanning by Trivy and SonarQube

Trivy was integrated with the Jenkins pipeline for scanning vulnerabilities in containers. It scans Docker images before deployment as shown in Fig 3.5. The tool detects vulnerabilities and proposes remediation. SonarQube is used to scan code for bugs, code smells, and security vulnerabilities to ensure that code quality is good and it follows security to pass into the next phase.

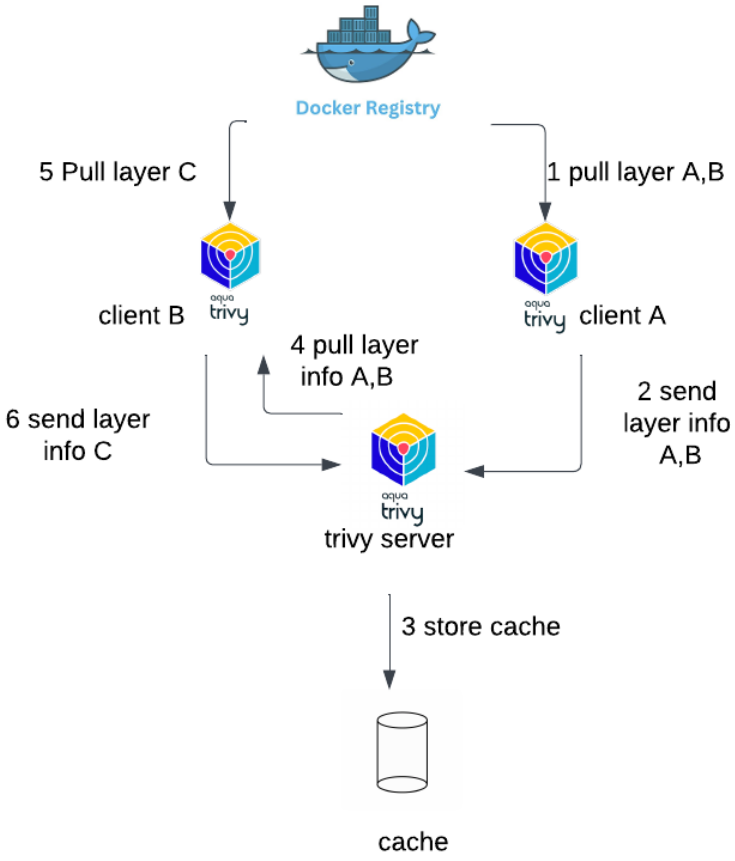

Fig. 3.5. Aqua Trivy: Vulnerability and Misconfiguration Scanning

3.4.2 Authentication and Authorization

Authentication and authorization have been configured within Jenkins so that only certain persons may edit the pipeline configurations as well as settings. Token was also generated for a secured access of the repos while the nexus usernames and password are being kept safe from which access into the artifact storage should be strict.

3.4.3 Email notifications and alerts

Extended email notifications to stakeholders are used so that extended notifications can let the stakeholders understand the important points within the programme- such as failed builds, successful deployment, or the presence of vulnerabilities. All these maintained transparency and permitted quick measures for avoiding additional downtime while preventing security compromise.

3.5. Monitoring and Feedback Systems

In short, Monitoring and feedback form the very core of a mature pipeline. Once the monitoring tool is integrated, the performance metrics of the pipeline with real-time alerts on anomalies would be provided.

3.5.1 Prometheus and Grafana for Monitoring

Prometheus monitored performance and server health, contributing to system metrics, resource usage and container statuses. Grafana did the data visualizations. So, real-time dashboards could be built with CPU usage and memory consumption, network traffic for Jenkins, Kubernetes, as well as deployed applications.

3.5.2 Blackbox and Node Exporter for System Metrics

The blackbox exporter was configured to report on the availability of a website, and the Node Exporter was configured to collect system metrics from all Jenkins nodes. The exporters were helpful in gathering granular details on the system to be proactive about issues before they become major pipeline-related problems.

3.6. Artifact Management and Deployment

Nexus was created as the artifact repository, where the build artifacts and Docker images were securely stored. Access controls were implemented in Nexus to manage the permissions. Only authorized users could upload or download the artifacts.

Jenkins was configured to publish the build artifacts to Nexus, so that there was a controlled environment for the storage and retrieval of artifacts.

3.6.1 Docker Image Tagging and Repository Management

They tag and push images into the Nexus repository, and thus, easily deploy; this way, versions were tracked and only the tested, approved images were deployed to the production environment.

3.7. Testing and Validation of Pipeline Performance

Testing was continuous throughout the pipeline implementation, with multiple regular tests on all aspects of the functioning and security of every component. Unit tests, integration tests, and end-to-end testing were automated using Jenkins to check code functionalities, point out bugs early, and all this provided prompt validation of performance issues.

3.7.1 Load Testing and Performance Validation

Load tests were run on the pipeline to establish how it would scale up based on workload fluctuations. It meant that the pipeline could withstand a full corporate setting with deployment and simultaneous building processes. This would help track performance metrics and point out the chokepoints, thus further optimizing the performance.

3.7.2 Security and Compliance Verification

Regular security verifications involving vulnerability scans, authentication checks, and RBAC testing were performed to ensure industry standards compliance for security. The outputs of these tests were incorporated into the adjustment of security policies so that the pipeline was kept resilient against emerging threats.

CHAPTER 4

IMPLEMENTATION

The implementation phase of this project involves setting up pipeline using a variety of DevOps tools and techniques. Each step in the implementation process is designed to ensure seamless integration, automated testing, and efficient deployment of applications in a scalable and secure environment.

The beginning of this phase starts by provisioning cloud infrastructure, along with the configuration of such tools as Kubernetes, Jenkins, Nexus, and SonarQube, which play a significant role in CI/CD. It also involves setting up a version control system, the management of containerized applications, and making sure that the code adheres to quality through automatic analysis tools. All steps have been executed and documented, thus giving a holistic overview of the process.

The following is the detailed step of each component in implementation.

4.1. First Stage: Creating an Ubuntu EC2 Instance in AWS

4.1.1. Sign in to the AWS Management Console

- Access the AWS Console at https://aws.amazon.com/console/ and log in with your AWS credentials.

4.1.2. Navigate to EC2

- In the AWS Console, search for "EC2" or go to **Services > EC2** under the "Compute" section.

4.1.3. Launch an Instance

- In the EC2 dashboard, click **Instances** from the sidebar and then **Launch Instance**.

4.1.4. Choose an Amazon Machine Image (AMI)

- Select **Ubuntu Server 20.04 LTS** (or your preferred version) from the list of AMIs and click **Select**.

4.1.5. Instance Type

- Select an appropriate instance type (e.g., t2.micro for lightweight workloads) and click **Next: Configure Instance Details**.

4.1.6. Instance Details

- Adjust network settings, subnets, and IAM roles as needed, or leave them as default. Click **Next: Add Storage**.

4.1.7. Storage

- Choose the root volume size and proceed to **Next: Add Tags**.

4.1.8. Add Tags

- Add tags for better organization (optional). Click **Next: Configure Security Group**.

4.1.9. Configure Security Group

- Allow SSH from your IP address. Optionally, open ports for HTTP / HTTPS based on your requirements.

4.2. Second Stage: Setting Up a Kubernetes Cluster (v1.28.1) Using kubeadm

Update System Packages (Run on both Master and Worker Nodes):

sudo apt-get update

4.2.1. Install Docker:

`sudo apt install docker.io -y`

Installs Docker, a container runtime necessary for running containerized workloads in Kubernetes.

`sudo chmod 666 /var/run/docker.sock`

Grants read, write, and execute permissions to all users for the Docker socket file, enabling non-root users to interact with the Docker daemon.

4.2.2. Install Kubernetes Dependencies:

`sudo apt-get install -y apt-transport-https ca-certificates curl gnupg`

Installs essential packages required to securely fetch and install Kubernetes packages from its repository.

`sudo mkdir -p -m 755 /etc/apt/keyrings`

Creates a directory to store GPG keys securely with appropriate permissions.

4.2.3. Add Kubernetes Repository and GPG Key:

`curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.28/deb/Release.key | sudo gpg --dearmor -o /etc/apt/keyrings/kubernetes-apt-keyring.gpg`

Fetches the GPG key for the Kubernetes package repository and stores it in the keyrings directory to ensure the authenticity of downloaded packages.

`echo 'deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] https://pkgs.k8s.io/core:/stable:/v1.28/deb/ /' | sudo tee /etc/apt/sources.list.d/kubernetes.list`

Adds the Kubernetes repository to the system's list of package sources.

4.2.4. Update Package List:

`sudo apt update`

Refreshes the package list to include the newly added Kubernetes repository.

4.2.5. Install Kubernetes Components:

`sudo apt install -y kubeadm=1.28.1-1.1 kubelet=1.28.1-1.1 kubectl=1.28.1-1.1`

Installs the specific version of Kubernetes components as shown in fig 4.2 and fig 4.3:

kubeadm: Used to bootstrap the cluster.

kubelet: Runs on all nodes and ensures that containers are running as expected.

kubectl: A command-line tool for interacting with the Kubernetes cluster.

Initialize Master Node:

`sudo kubeadm init --pod-network-cidr=10.244.0.0/16`

Initializes the Kubernetes control plane on the master node and specifies the Pod network CIDR (Classless Inter-Domain Routing) required for the networking solution as shown in fig 4.1.

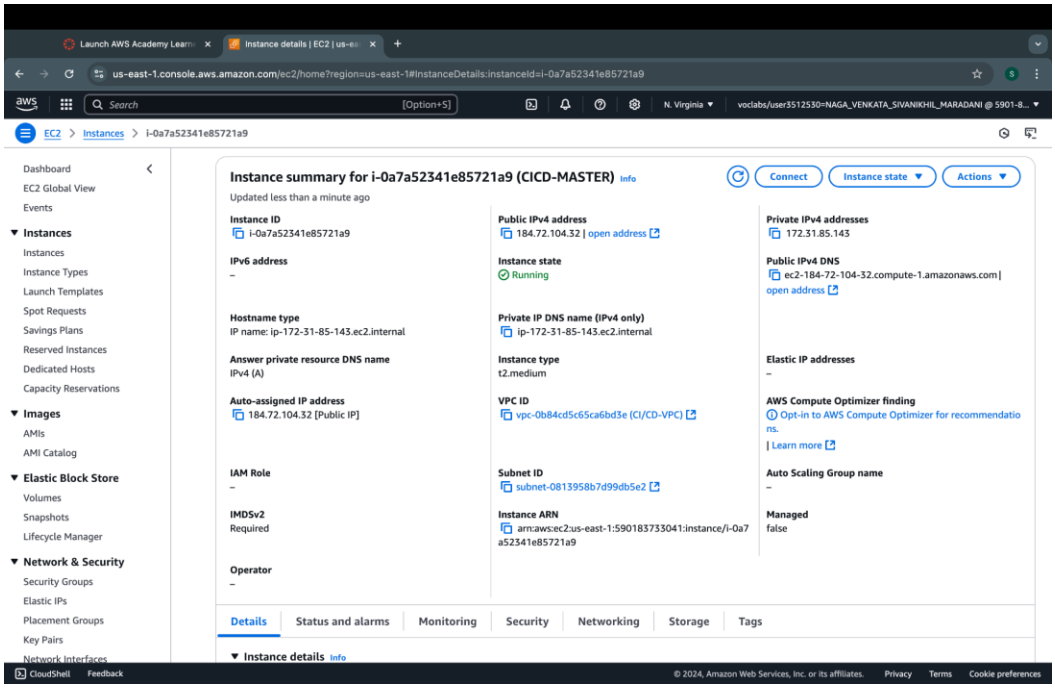

Fig. 4.1. Creating Master node Instance

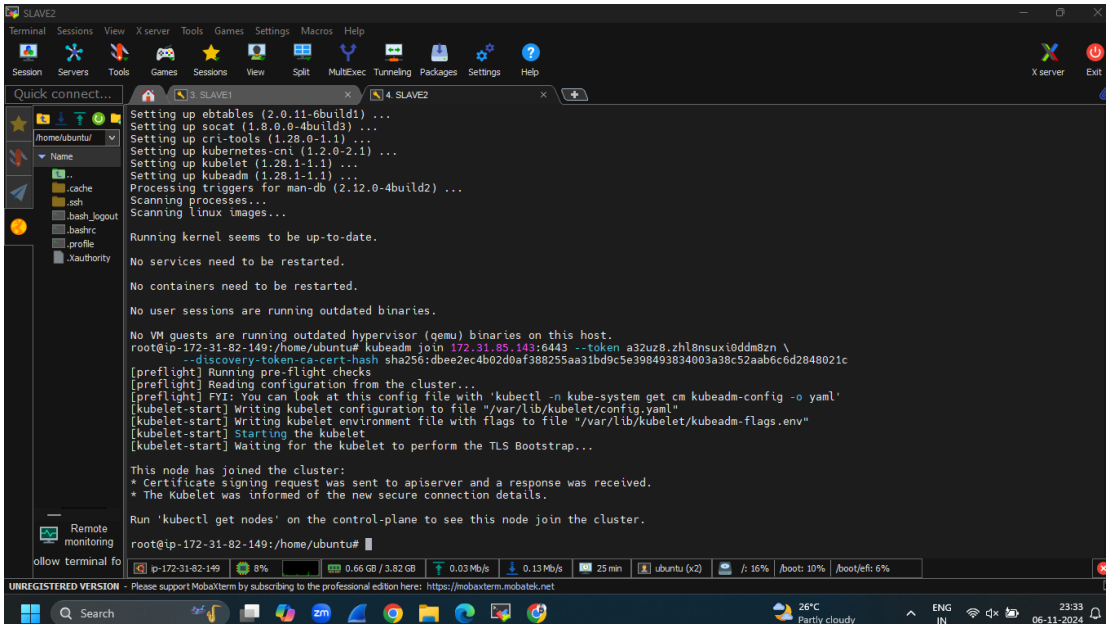

Fig. 4.2. Connecting Slave nodes 1 to master node

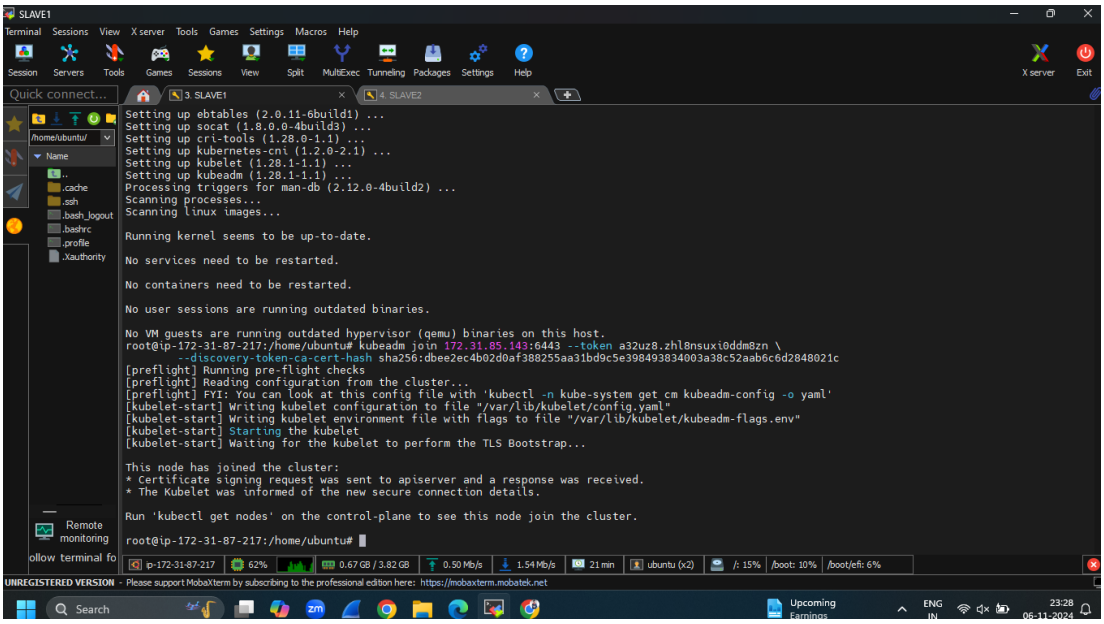

Fig. 4.3. Connecting Slave nodes 2 to master node

4.2.6. Configure Kubernetes Cluster:

`mkdir -p $HOME/.kube`

Creates a directory to store Kubernetes configuration files for the current user.

`sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config`

Copies the admin configuration file to the user's kube directory to allow kubectl to communicate with the cluster.

`sudo chown $(id -u):$(id -g) $HOME/.kube/config`

Changes ownership of the configuration file to the current user as shown in fig 4.5.

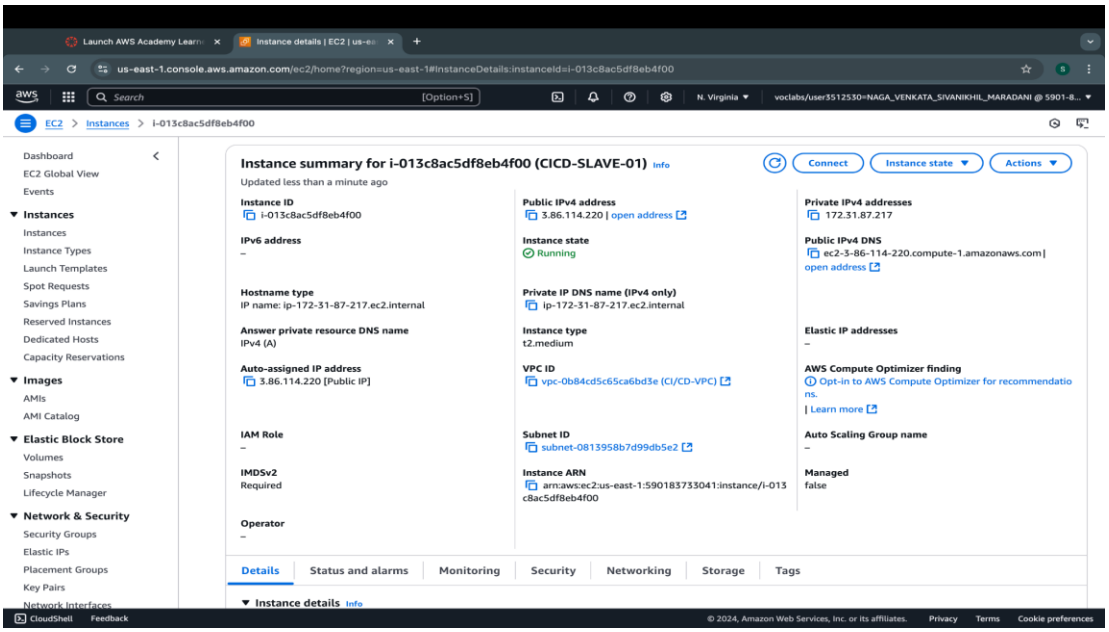

Fig. 4.4. Creating Slave-01 instance

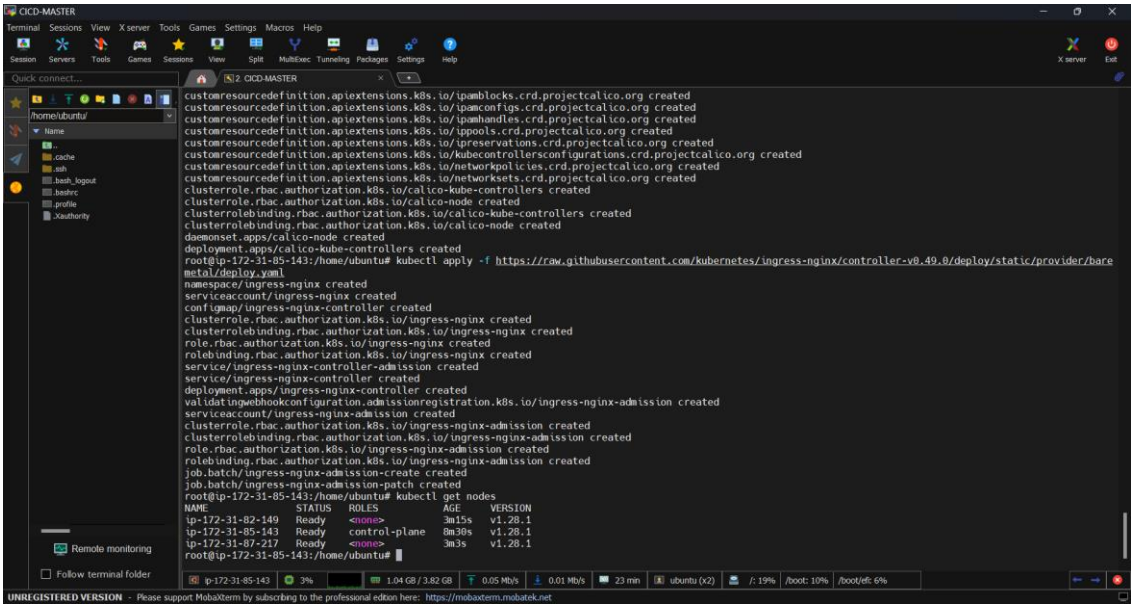

Fig. 4.5. Checking the nodes are connected to master after the configuration

4.2.7. Deploy Calico Networking Solution:

`kubectl apply -f https://docs.projectcalico.org/manifests/calico.yaml`

Deploys Calico, a popular networking and network policy solution for Kubernetes. This step ensures Pods can communicate with each other across the cluster.

4.2.8. Deploy NGINX Ingress Controller:

`kubectl apply -f ` `https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v0.49.0/deploy/static/provider/baremetal/deploy.yaml`

Deploys the NGINX Ingress Controller, which manages external access to services within the cluster using HTTP/HTTPS routes.

4.3. Third Stage: Installing Jenkins on Ubuntu

Save the following script as install_jenkins.sh:

4.3.1. Install OpenJDK 17 JRE

`sudo apt install openjdk-17-jre-headless -y`

Installs OpenJDK 17 JRE for Java Runtime Environment in headless mode, which omits graphical components.
Java is a requirement for running Jenkins, and OpenJDK 17 is suggested for the latest Jenkins version
The -y flag autoconfirms the installation prompt.

4.3.2. Download Jenkins GPG key

`sudo wget -O /usr/share/keyrings/jenkins-keyring.asc https://pkg.jenkins.io/debian-stable/jenkins.io-2023.key`

Downloads the Jenkins GPG key from the official Jenkins repository and saves it at /usr/share/keyrings for security. The GPG key is what will be used to check the authenticity and integrity of the Jenkins package.

4.3.4. Add Jenkins repository

`echo deb [signed-by=/usr/share/keyrings/jenkins-keyring.asc] https://pkg.jenkins.io/debian-stable binary/ | sudo tee /etc/apt/sources.list.d/jenkins.list > /dev/null`

4.3.5. Adds the Jenkins Debian repository to the system's package sources list.

The signed-by option specifies the location of the GPG key, ensuring only packages signed with the Jenkins key are trusted.
The tee command writes the repository configuration to the file /etc/apt/sources.list.d/jenkins.list.

4.3.6. Update package repositories

`sudo apt-get update`

Updates the package index to include the newly added Jenkins repository, making the Jenkins package available for installation.

4.3.7. Install Jenkins

`sudo apt-get install jenkins -y`

Installs the latest stable version of Jenkins from the repository.

The -y flag confirms the installation automatically.

Make the script executable and run it:

`chmod +x install_jenkins.sh`

Changes the file permissions to make the script executable. This will enable the script to be run directly using./install_jenkins.sh.

`./install_jenkins.sh`

Runs the script to automate the whole process of installing OpenJDK, adding the Jenkins repository, and installing Jenkins itself as shown in fig 4.6.

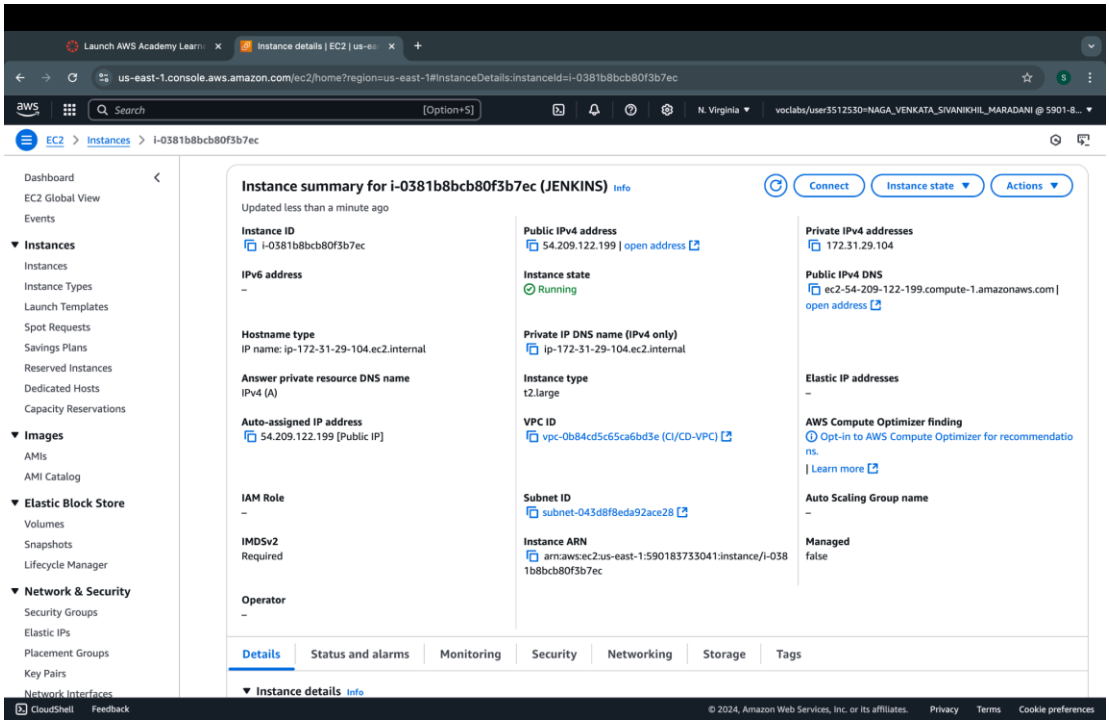

Fig. 4.6. Creating Instance for Jenkins Server

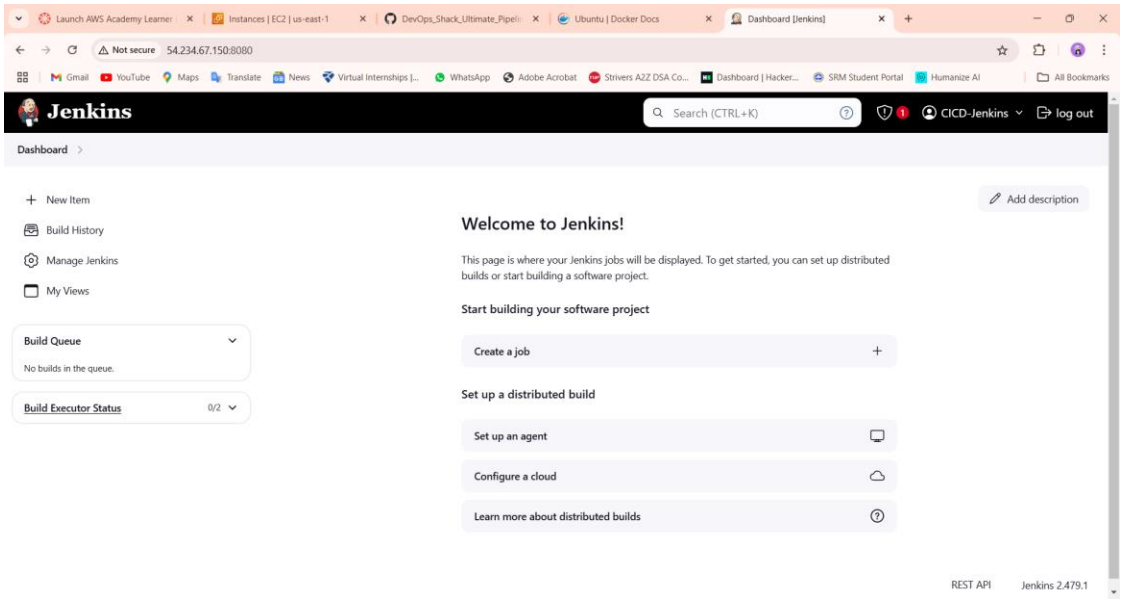

Fig. 4.7. Exposing Jenkins Dashboard On Port:8080

4.4. Fourth Stage: Setting Up Nexus Repository Manager

4.4.1. Install Docker

Save the following script as install_docker.sh:

4.4.2. Update package repositories

`sudo apt-get update`

4.4.3. Install required dependencies

`sudo apt-get install -y ca-certificates curl`

4.4.4. Create directory for Docker GPG key

`sudo install -m 0755 -d /etc/apt/keyrings`

4.4.5 Download Docker's GPG key

`sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o /etc/apt/keyrings/docker.asc`

4.4.6. Set proper permissions for the GPG key

`sudo chmod a+r /etc/apt/keyrings/docker.asc`

4.4.7. Add Docker repository to Apt sources

*`echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc] https://download.docker.com/linux/ubuntu *

$(. /etc/os-release && echo "$VERSION_CODENAME") stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null`

4.4.8. Update package repositories and install Docker

`sudo apt-get update`

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

4.4.9. Make the script executable and run it:

`chmod +x install_docker.sh`

`./install_docker.sh`

4.4.10. Create and Run a Nexus Container

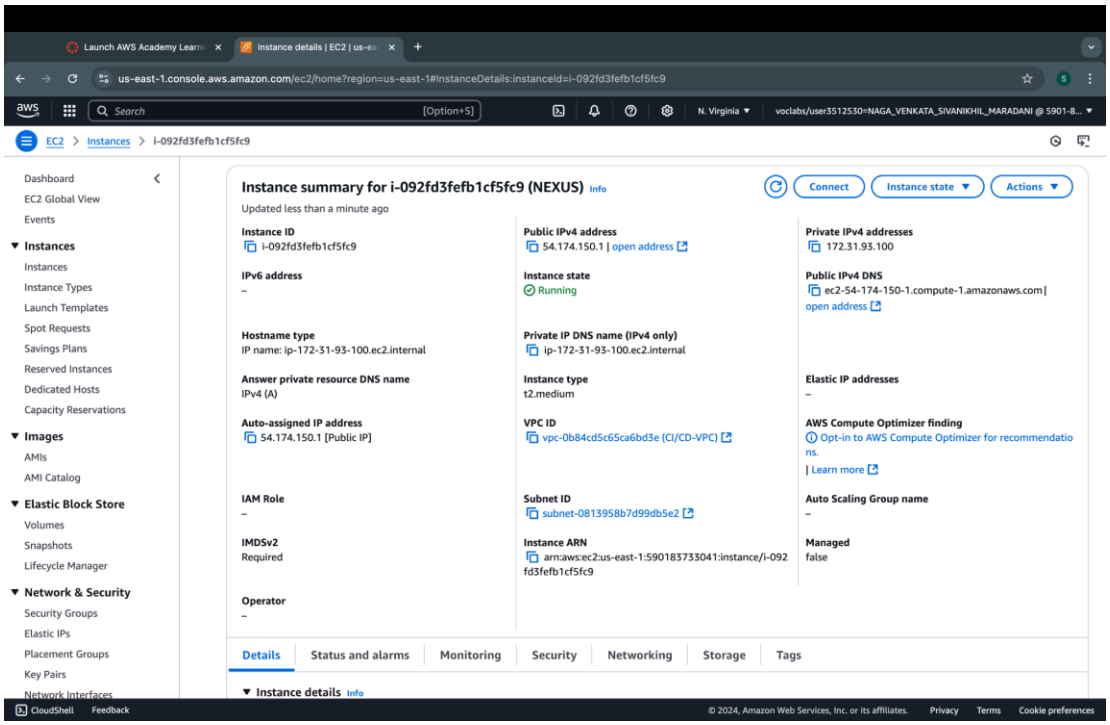

Fig. 4.8. Creating Instance For Nexus Server

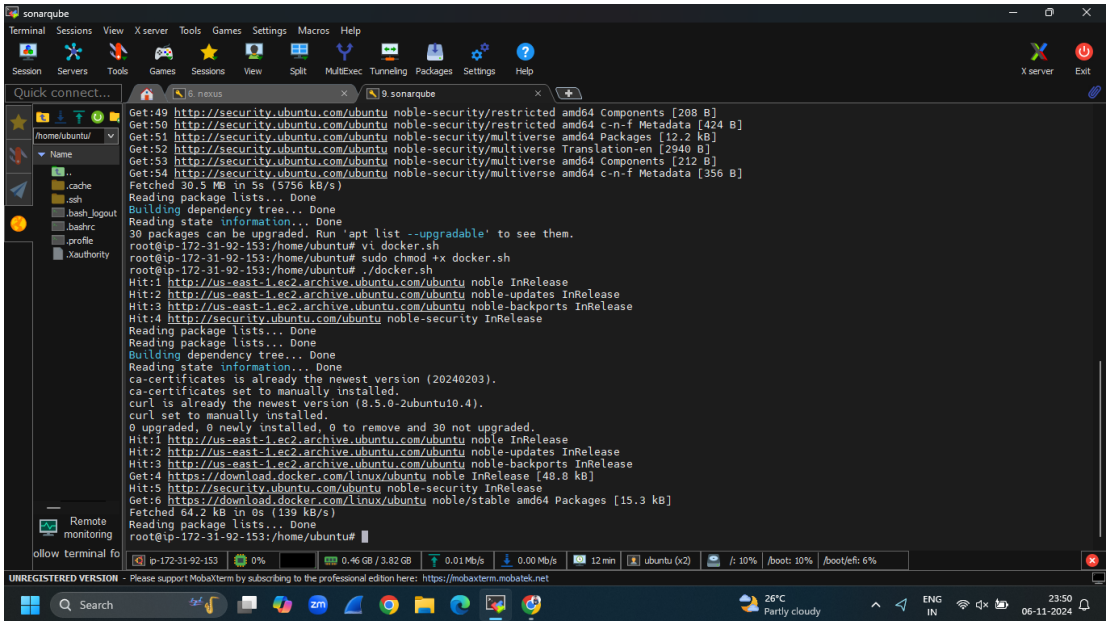

Fig. 4.9. Installing and Running the Nexus

Run the following command to create a Nexus container:

`docker run -d --name nexus -p 8081:8081 sonatype/nexus3:latest`

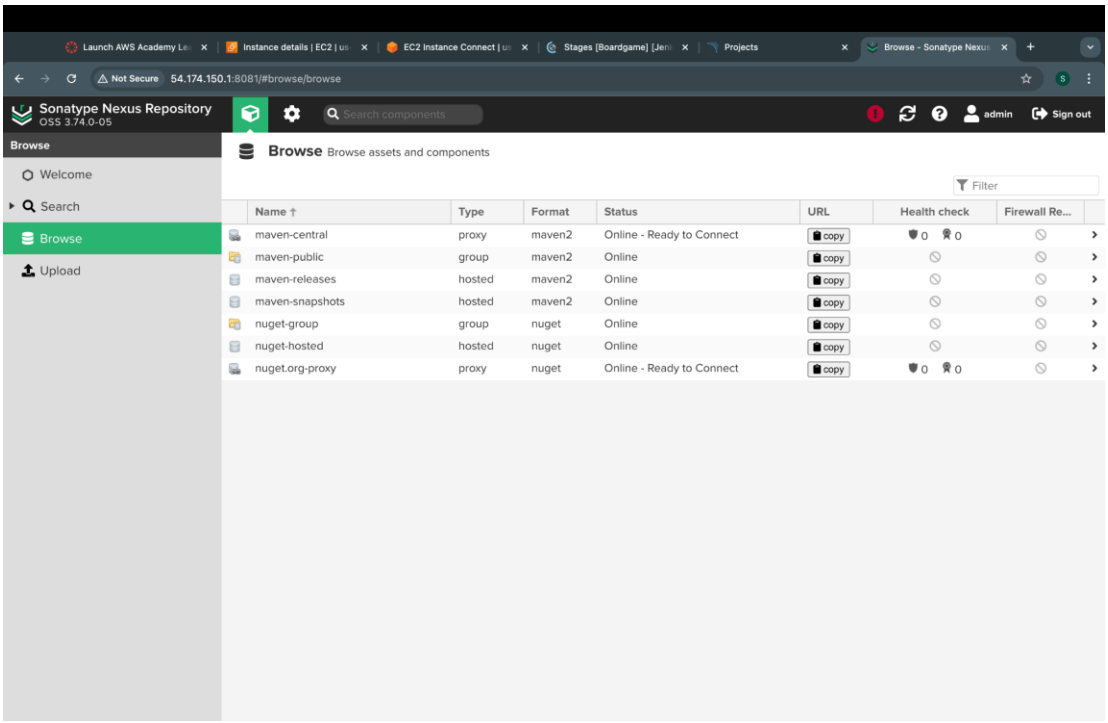

Fig. 4.10. Nexus Repository assets and components

4.4.11. Retrieve Initial Nexus Password

Get the container ID:
`docker ps`

- **Navigate to the password file:**
  `cd sonatype-work/nexus3`

  cat admin.password

- **Exit the container shell:**
  `Exit`

4.5 Fifth Stage: Setting Up SonarQube

4.5.1. Install Docker

If Docker is not already installed, follow the same steps from the **Fourth Stage**.

4.5.2. Create and Run a SonarQube Container

Run the following command to set up SonarQube:

`docker run -d --name sonar -p 9000:9000 sonarqube:lts-community`

- **Options Explained**:
  - --name sonar: Name the container as "sonar".
  - -p 9000:9000: Map port 9000 of the host to port 9000 of the container.

4.5.3. Access SonarQube

Navigate to:
`http://<VM_IP>:9000`

- Replace <VM_IP> with the address of the machine hosting the Docker container as shown in fig 4.11.

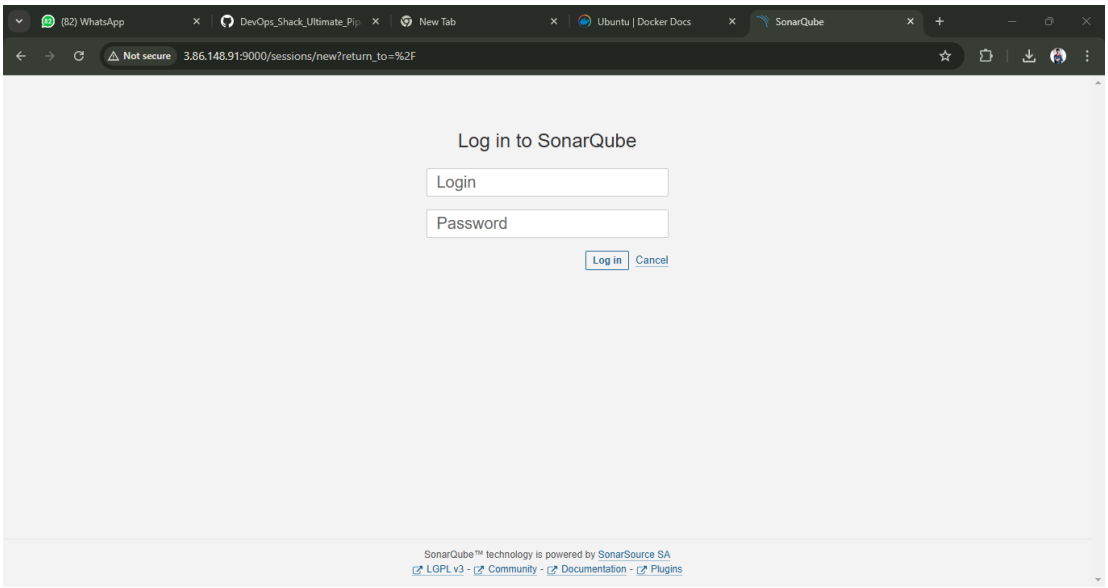

Fig. 4.11. Exposing SonarQube Dashboard on Port:9000

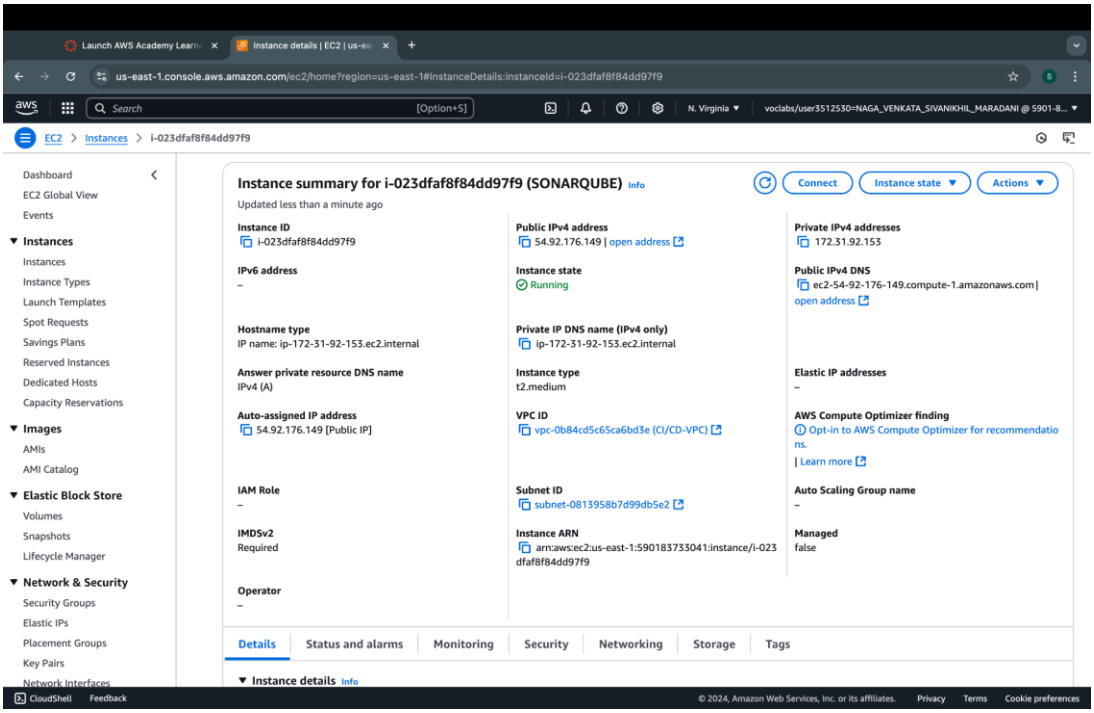

Fig. 4.12. Creating Instance for Sonarqube Server

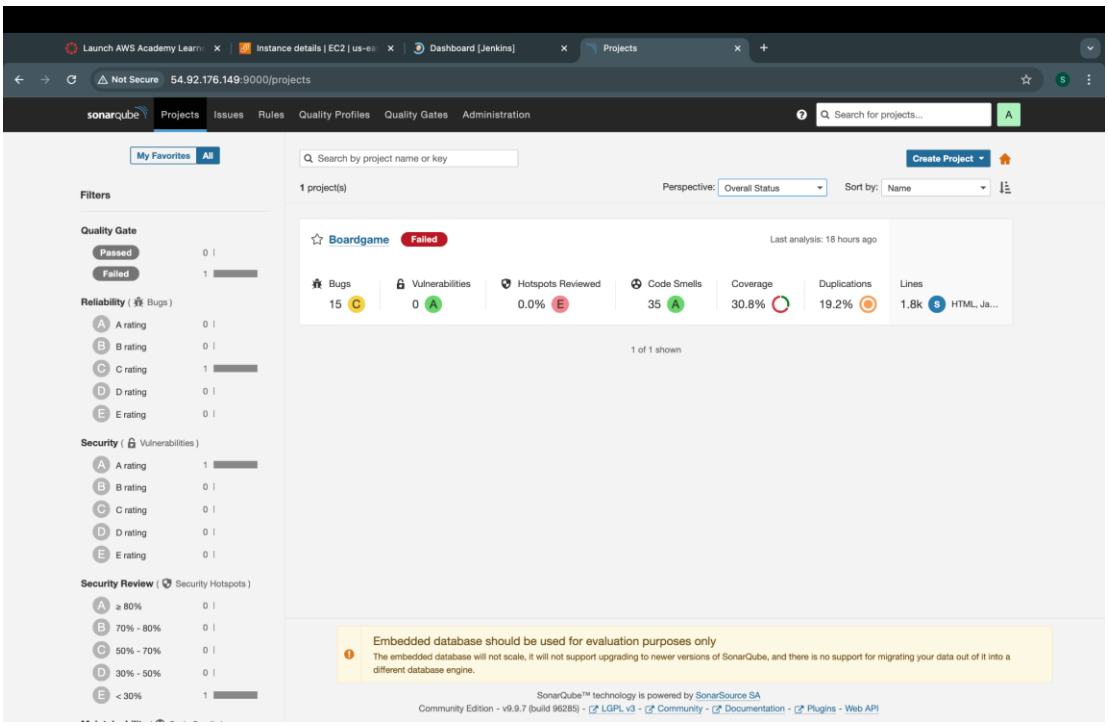

Fig. 4.13. SonarQube Dashboard

4.6. Sixth Stage: Steps to Create a Private Git Repository and Push Code

4.6.1. Create a Private Git Repository

- Go to your preferred Git hosting platform (e.g., GitHub, GitLab, or Bitbucket).
- Log in to your account or sign up.
- Create a repository:
    - Provide a repository name.
    - Make the repository visible to Only you.
    - Create New repository.

4.6.2. Create Personal Access Token

- Go to account settings or profile settings.
- In Developer settings or PAT, click the new token.
- Generate personal access token:
    - click to select repo, the access to this repository and some others copy your freshly created token; it won't appear anywhere again.

4.6.3. Clone the Repository on your machine

- Open your Git-Bash or any other terminal.
    - To get to the folder to clone the repository, go:
      `cd /path/to/your/directory`
- Clone the repository with this command:
  `git clone <repository_URL>`
- Substitute <repository_URL> for the URL of your private repository.

4.6.4. Add Your Source Code Files

Change to the cloned repository directory
`cd <repository_name>`

Add your source code files here, or create some.

4.6.5. Stage and Commit Changes

using the git add command
`git add .`

Commit the changes with a meaningful message:
`git commit -m "Initial commit"`

4.6.6. Push Changes to the Repository

Push into the Repository:
git push

- If it's your first time pushing, specify the remote and branch:
  `git push -u origin master`
- Replace master with the branch name if necessary.

4.6.7. Authenticate Using a Personal Access Token

When prompted for credentials:

- Enter your username (usually your email).
- Use your personal access token as the password.

By following these steps, you can securely create and manage a private Git repository, connect to it locally, and push your code changes using a personal access token.

4.7. Installing Plugins in Jenkins

Jenkins provides a robust plugin ecosystem to extend its functionality. Below is a guide to installing essential plugins for various development and deployment needs:

4.7.1. Eclipse Temurin Installer

Purpose: Automatically installs and configures the Eclipse Temurin JDK (formerly AdoptOpenJDK).
Steps to Install:
Search for Eclipse Temurin Installer and select it.
Click Install without restart.

4.7.2. Pipeline Maven Integration

Purpose: Allows Maven support for Jenkins Pipelines, allowing direct usage of Maven commands in Pipeline scripts.
Steps to Install:
Follow the same process as above.
Search for Pipeline Maven Integration and install it.

4.7.3. Config File Provider

Purpose: Allows centralized definition of configuration files (e.g., properties, XML, JSON) for Jenkins jobs.
Steps to Install:

Search for Config File Provider in the plugin manager.
Click Install without restart.

4.7.4. SonarQube Scanner

Purpose: It will connect Jenkins with SonarQube, enabling code quality and security analysis.
Steps to Install:-
Search for SonarQube Scanner in the plugin manager.
Install it similarly

4.7.5. Kubernetes CLI

Purpose: It makes possible for Jenkins to communicate with the Kubernetes clusters by use of the Kubernetes CLI kubectl.
Steps to install:
Search for Kubernetes CLI in the plugin manager
Install it without restarting the Jenkins

4.7.6. Kubernetes Plugin

Purpose: Integrates Jenkins with Kubernetes to run Jenkins agents as Kubernetes pods, providing dynamic scaling and efficient resource use.
Steps to Install:
Search for Kubernetes in the plugin manager.
Install using the default process.

4.7.7. Docker Plugin

Purpose: It enables interaction with Docker so that Jenkins will be able to build images, run containers, and manage the Docker registries.
Steps to Install:
Search for Docker in the plugin manager.
Click Install without restart.

4.7.8. Docker Pipeline Plugin

Purpose: Adds steps to build, publish, and manage Docker containers to Jenkins Pipeline through Pipeline scripts.
Installation Steps
Search for Docker Pipeline in the update center of your Jenkins.
Install like other plugins.
Post Installation Configuration as shown in fig 4.14.

After installing the plugins,

Credentials: Add credentials as necessary for each plugin. This can be, for example, SonarQube tokens, Kubernetes config, or Docker credentials.

Global Configuration: Access through Manage Jenkins → Global Tool Configuration or relevant settings for paths and options of each plugin.

Job-Level Configuration: Configure the plugin-specific settings in a particular Jenkins job or pipeline as necessary.

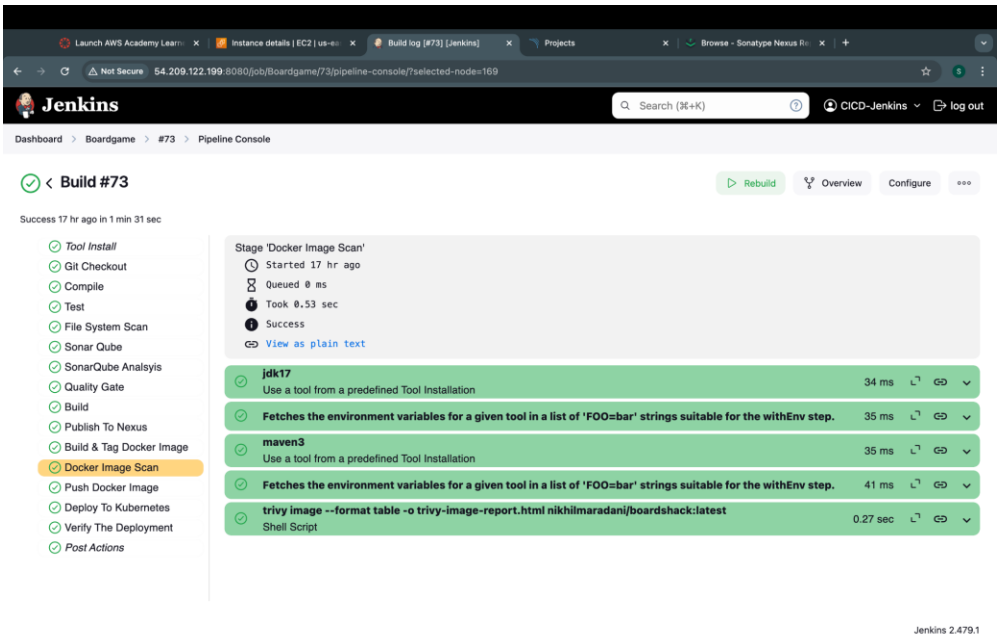

Fig. 4.14. Jenkins Pipeline Console to build and test the pipeline

CHAPTER 5

RESULTS AND DISCUSSIONS

Results from the metrics and visualization generated for CPU utilization, network traffic, HTTP durations, and probe response time of AWS EC2 instances. The discussion provides a deep dive into the results as well as implications on system performance and ways to optimize.

5.1. CPU Utilization of AWS Instances

The CPU utilization metrics on the three AWS EC2 instances of Master, Slave 1 and Slave 2 indicate that they are stable and efficient under light to moderate workloads.

5.1.1 Instance Usage Trends:

All displays have relatively low CPU usage, mainly fluctuating below 12.6% as shown in fig 5.1. Low utilization rates point to the ability of the system to manage its current workloads without undue stress. From time to time, though, spikes were observed in CPU usage to around 25.1%, presumably due to short-lived transient workloads, as in the case of batch jobs or major computation bursts. While these spikes are momentarily noticeable, they do not persist; this consequently supports that such transient demands are managed well by the infrastructure.

5.1.2. Load Distribution:

The baseline utilization varied slightly with different usage patterns, but remained consistent across all instances, which means the workload was well balanced. This indicates that the system load-balancing mechanisms are efficiently distributing load across instances, while little deviations would occur because of different processes within instances.

5.1.3. Performance Insights:

The low and stable CPU utilization indicates the AWS infrastructure is still quite scalable for its workloads. The short, sharp spikes further confirm the system's performance will not degrade under bursty workloads. These results also demonstrate that the instances utilized are cost-effective while remaining capable and prepared for a potential increase in workload in the future.

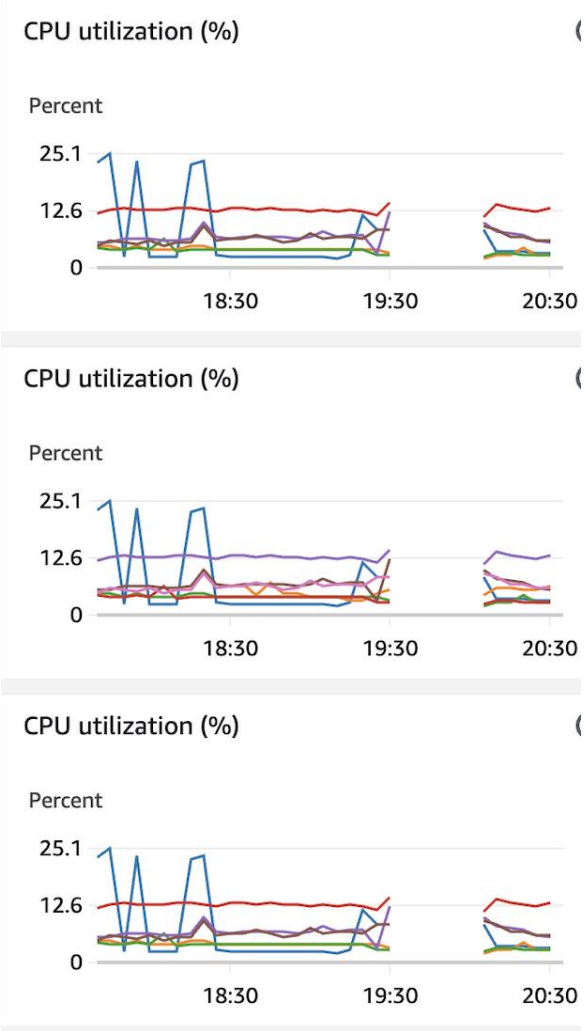

Fig. 5.1.CPU Utilization

5.2. Network Out Utilization of AWS Instances

The Out metrics of the Network shows instances' behavior regarding outgoing traffic.

5.2.1. Traffic Patterns:

The out traffic for all 3 instances of Master, Slave 1 and Slave 2 reflected initial increases to around 19.3 MB and then fell afterwards as shown in Fig 5.2. All these spikes occurred within a short time window most probably due to batch data transfer, deployment events, or a heavy API request burst. Beyond these peaks, the networks remained stable and stayed well below 9.64 MB for the remainder of the observation time.

5.2.2. Instance Comparisons:

Quite similar traffic patterns were observed between instances with minor differences in the volume of outgoing traffics at later times. These fluctuations might relate to application-specific loads or small variations in workload execution.

5.2.3. Performance Inferences:

It managed to successfully absorb high-intensity bursts of short-lived nature, for data-intensive or network-intensive applications. Stability after the bursts in traffic patterns reflects good utilization of network resources. These observations guarantee that the AWS instances handle workload with fluctuating network demands and remain stable under such a scenario.

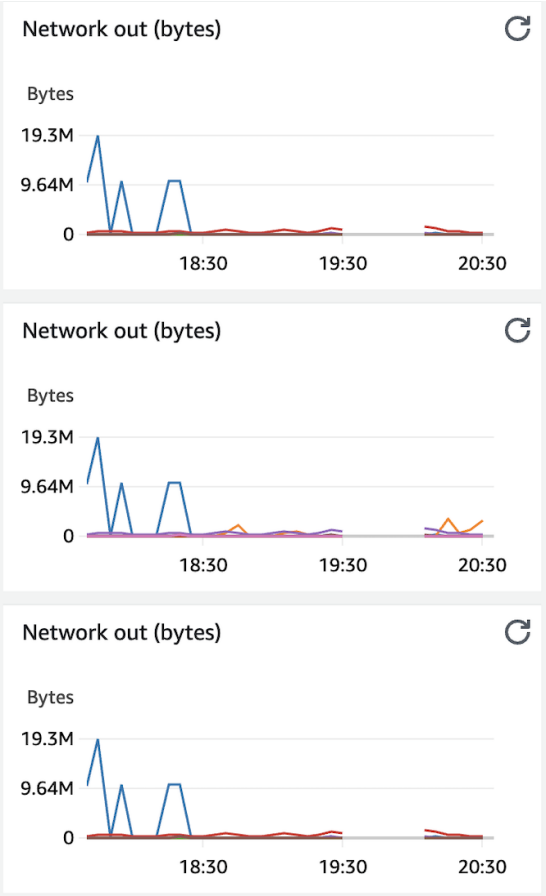

Fig. 5.2. Network Out Utilization

5.3 Network In Utilization Across AWS Instances

The network patterns of inbound traffic to the AWS EC2 instances indicate how dynamic the inflow of data.

5.3.1. Traffic Spikes:

There was a noticeable peak in all 3 instances of Master, Slave 1 and Slave 2 for inbound traffic at approximately 18:30 across all instances as shown in Fig 5.3. A scheduled event such as batch data synchronization, application updates, or heavy inbound requests might have caused it.

5.3.2. Declining Trends:

After the peak, traffic levels dropped to a minimum, indicating a state of idleness or steady state. The behavior indicates the system has the potential to revert rapidly back to baseline activity after handling peak loads.

5.3.3. Instance Comparison:

Even though all instances showed similar patterns of spikes, slight differences in post-spike traffic were observed, which may be due to differences in workload processing or to the application configurations. These differences are important because they reflect how well a system can dynamically allocate resources based on demand.

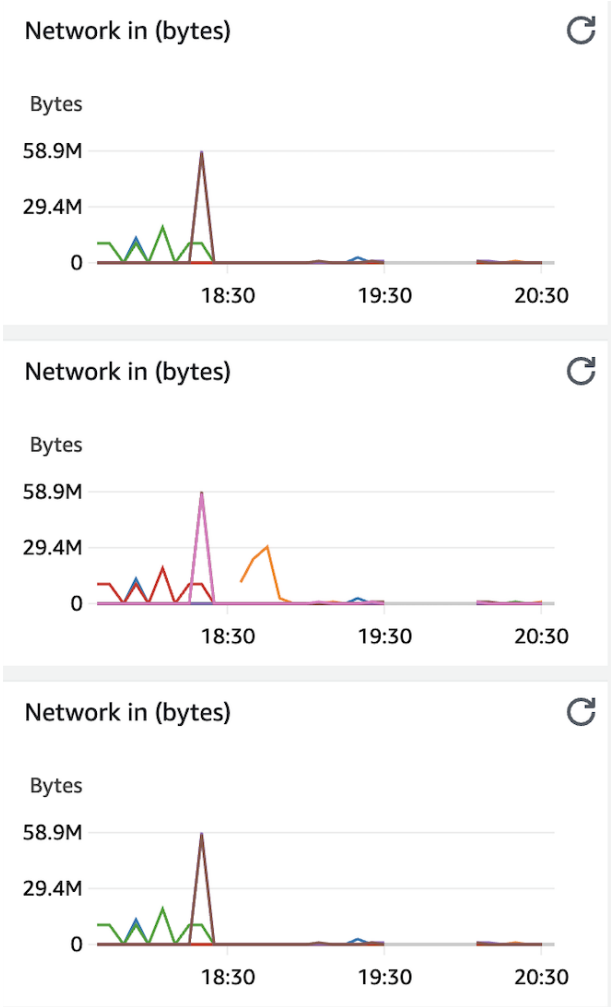

Fig. 5.3. Network In Utilization

5.4. Network Packets Analysis

The "Network Packets In" metric captures the point of detail network activity in terms of packet counts across the instances.

5.4.1. Traffic Patterns:

All 3 instances of Master, Slave 1 and Slave 2 peaked sharply in terms of packet counts from 18:30 as shown in Fig 5.4, similar to peak trends shown in Network In metrics. Such a high volume of processes may involve API requests, data synchronization or scheduled application tasks.

5.4.2. Post-Spike Stability:

After the initial surge, packet counts stabilized at very low levels, showing little activity on the network. The stability after the surge reflects how the system adapts to fluctuating demands while still maintaining optimal performance as shown in 5.5.

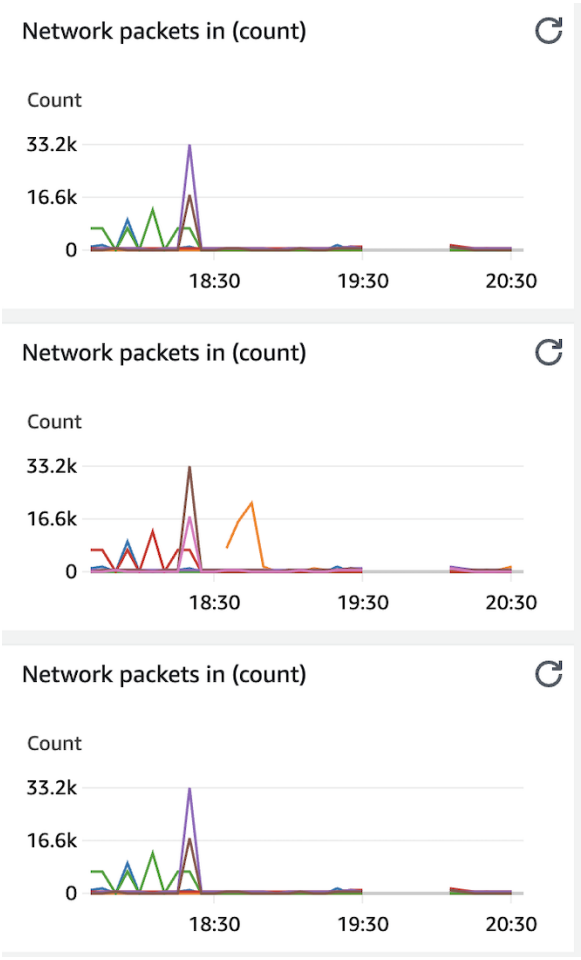

Fig. 5.4. Network Packets

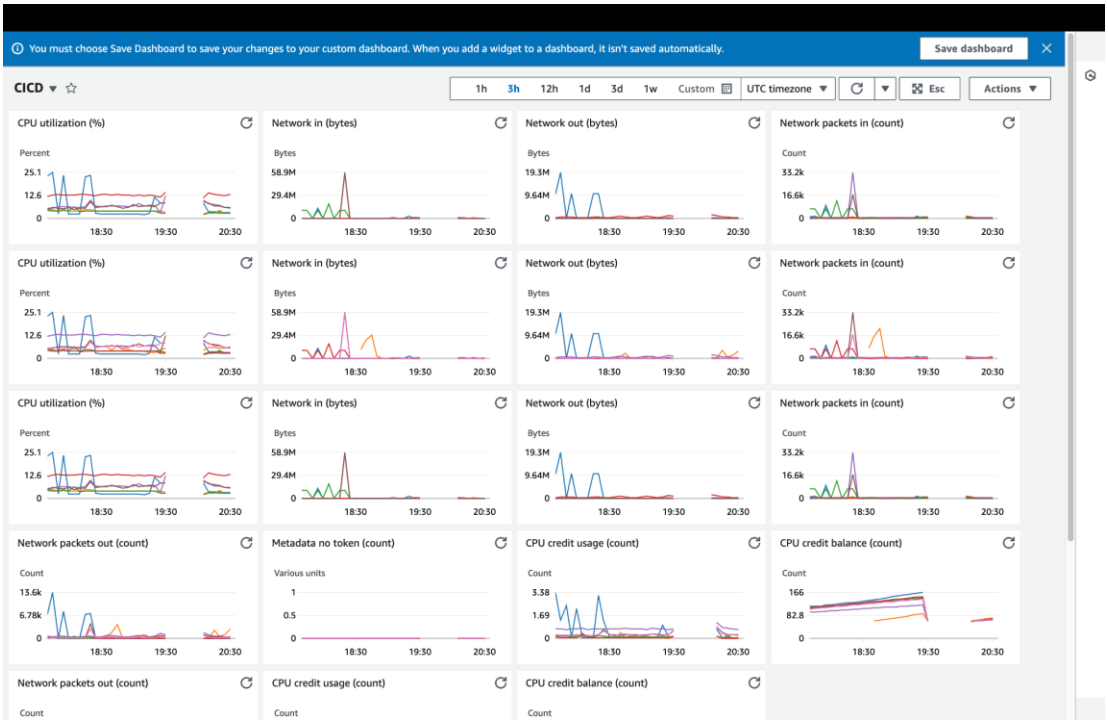

Fig. 5.5. AWS instances infrastructure analysis of Master, slave 1 and slave 2 nodes

5.4.3. Instance Behaviour:

There were minor fluctuations in spike intensity and time, presumably related to differences in workload characteristics or instance-specific settings. One instance showed a second, smaller spike following the dominant peak, which might indicate other processing requirements specific to that instance.

5.4.4. Implications:

It reveals the need to monitor packet counts as well as data volume for effective resource management. It in any case reconfirms the fact that the system is able to handle peak traffic efficiently and keeps intact even at idle periods.

5.5. HTTP Duration Analysis

The analysis of HTTP durations captures the performance of various stages of HTTP operations, which includes connection, processing, and transfer times.

5.5.1. Observations Stage-Wise

Processing times (yellow) dominated total HTTP time, as most of the delay occurred between server-side operation and network latency or connection setup. The connect time (green) and resolve time (orange) were uniformly low, which suggests little waiting during connection establishment and resolving DNS. TLS (red) and transfer times were zero or negligible, which emphasized minimal overhead of encryption and efficient data transfer.

5.5.2. Performance Spikes:

Periodic spikes in processing time, observed around 00:43:00 and 00:45:00, point to times with higher workload or resource contention in these parts of the timeline as shown in Fig 5.6. These points, consequently, mark potential inspection areas for server-side operations.

5.5.3. Insights:

The consistent performance of the metrics of lower durations reflects an optimized usage of AWS's networking abilities. Deeper analysis of processing times may identify potential spots for boosting application-level efficiency, particularly at peak activity.

Fig. 5.6. HTTP Duration Analysis

5.6. Probe Duration Analysis

Probe duration metrics capture the time at which responses are received from those endpoints that have been monitored across AWS infrastructure.

5.6.1. Endpoint Analysis

The green endpoint had periodic patterns with high latency nearly every 10 seconds as shown in Fig 5.7, changing to near-zero values. These patterns suggest potential issues such as inefficient load balancing or periodic tasks affecting time to respond. The yellow endpoint demonstrated very stable response times; the endpoint was reliable and consistent.

5.6.2. Periodic Patterns:

The periodic fluctuations around the green endpoint probe time point to possible opportunities to optimize. The zero intervals may be indicative of idle times or service outage and thus merit further analysis.

5.6.3. Inferences

Analysis of probe time conveys the significance of monitoring the performance of the endpoints. Optimized response times of the green endpoint will enhance the overall reliability of the system, especially in cyclic workloads as shown in Fig 5.8.

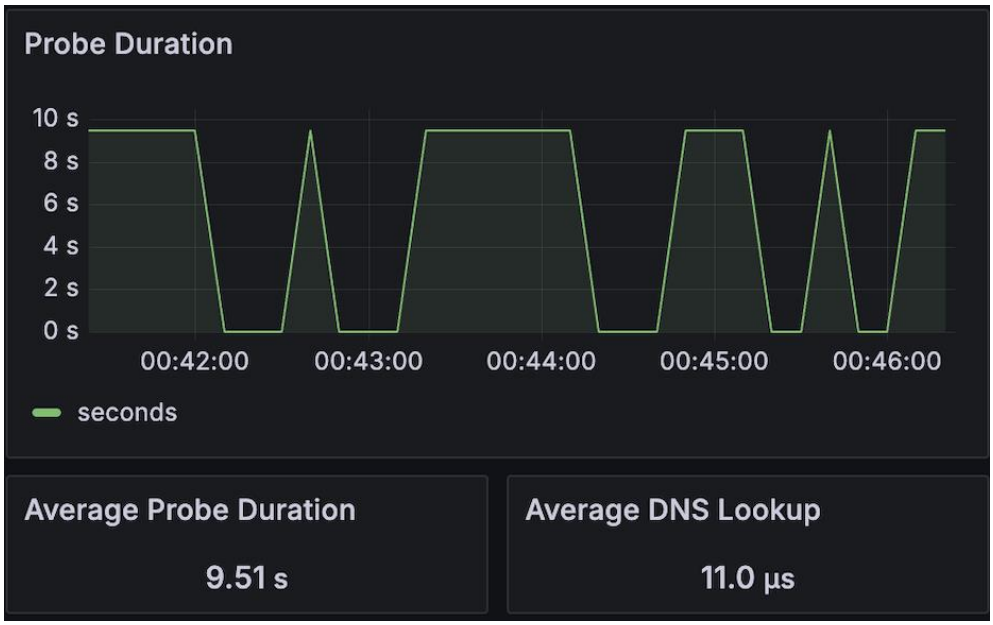

Fig. 5.7. Probe Duration Analysis 1

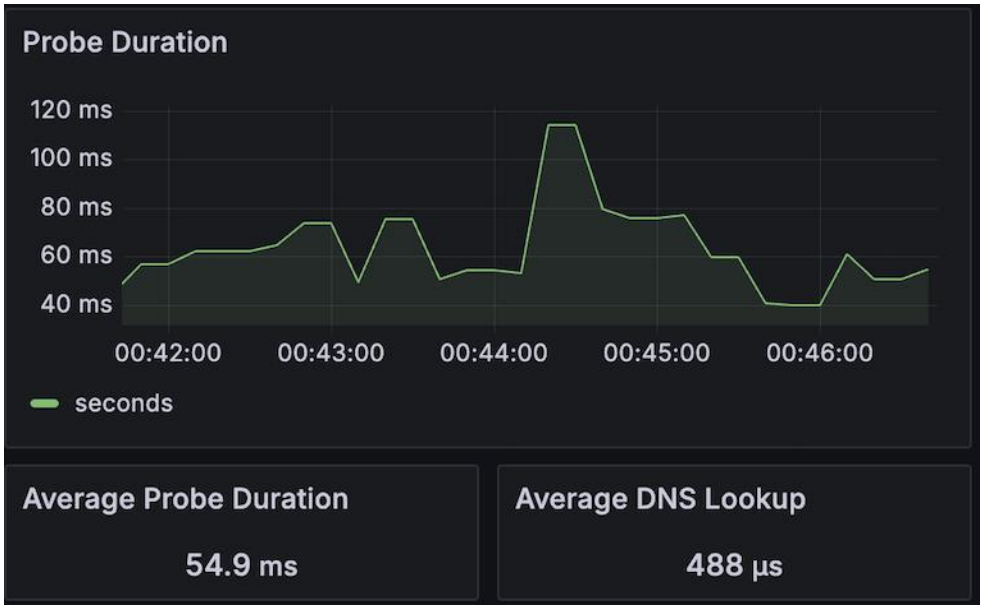

Fig. 5.8. Probe Duration Analysis 2

5.7. Global Insights and Recommendations

The metrics and visualizations will thus give a full overview of how the AWS infrastructure performs in relation to its CPU, network, HTTP operations, and probe durations. Key Takeaways:

5.7.1. Performance Optimization

Probe the root causes of periodic CPU utilization, network traffic, and probe durations. Optimize server-side operation times for lower processing times during peak HTTP operation as shown in Fig 5.9.

5.7.2. Scaling and Resilience

Leverage auto-scaling mechanisms to efficiently handle workload spikes. Ensure instances are provisioned to handle both short-term bursts and sustained workloads.

5.7.3. Security Enhancements:

Though TLS and encryption metrics were minor, the deployment of HTTPS for all communications ensures a secure transmission of data.

5.7.4. Advanced Monitoring

Use Grafana to visualize performance trends in real-time Application logs correlated with network and CPU metrics for actionable insights.

Fig. 5.9. Global Probe Duration

The metrics collected from Prometheus are critical appraisal of the efficiency, reliability, and scalability of pipeline. This discussion enlarges on insights from data gathering, its implications, and opportunities for improvement.

5.8. Evaluation of Networking Efficiency

Prometheus logs expose a number of performance-oriented metrics for how effectively networking operations are being carried out in the pipeline:

5.8.1. DNS Lookup Time:

The DNS resolution time stands at 0.000012185 seconds, which means that the resolution is pretty optimized. Optimized DNS resolution is very important in a modern CI/CD pipeline because a lot of integrations done externally depend on service lookups, thus inducing delays.

5.8.2. Connection and Processing Times:

The connection phase took 0.001711466 seconds. The request processing phase took 0.007291979 seconds. These statistics prove the network and server substructure of the system works within stringent tolerance limits.

5.8.3. Total Probe Duration:

The overall duration of the probe is 0.009893655 seconds, proving efficiency in the handling of incoming requests. This further finds its affirmation in streamlined routing, low latency, and powerful server configurations.

5.9. HTTP Metrics:

Indicators of Application Health

The HTTP-related metrics expose the following concerns over the responsiveness and correctness of the application in table 5.1:

5.9.1. Response Content:

Probe http content length is marked as -1, which actually means that it does not know a content length. This information is not useful for pipeline operations, but still making sure that HTTP responses return all information by headers improves debugging capability and functionality with downstream systems.

5.9.2. HTTP Status Code:

A status code of 200 will ensure that the target endpoint for the pipeline is good and responding to probes without any downtime. This ensures high availability, which is important for production-grade CI/CD.

Table 5.1. Metrics from Prometheus Logs

Metric	Value
Probe dns lookup time seconds	0.000012185
Probe duration seconds	0.009893655
Probe failed due to regex	0
Probe http content length	-1
Probe http duration seconds	See below
- {phase="connect"}	0.001711466
- {phase="processing"}	0.007291979
- {phase="resolve"}	0.000012185
- {phase="tls"}	0

- {phase="transfer"}	0.00057556
Probe http redirects	0
Probe http ssl	0
Probe http status code	200
Probe http uncompressed body length	3843
Probe http version	1.1
Probe IP add hash	682023683
Probe IP protocol	4
Probe success	1

5.9.3. Uncompressed Body Length over HTTP:

The body length amounts to 3843 bytes. This post has a large payload, meaning it's worthwhile to send in the probe. It also indicates that the services can handle payloads of this size.

5.10. Security Observations

One of the key takeaways from Prometheus is that there is no encryption by TLS during communication. The probe http ssl value is 0, which means that it did not use the secure protocols in the communication. This puts it into a security risk, particularly when dealing with sensitive data or interacting with external systems. While the system shows strong performance, implementing all communications through HTTPS is essential to protect:

- **Data Integrity:** From tampering at transmission.
- **Confidentiality:** Protecting Credentials and Configuration Data.
- **Compliance:** Compliance with industry standards on secure communication.

5.11. Protocol and IP Handling

The IP protocol used during the probe is IPv4, as indicated by the probe IP protocol metric being 4. IPv4 is very widely supported, but IPv6 is becoming rapidly adopted. Therefore, there is a need to support dual-stack configurations in pipelines nowadays for future scalability and compatibility in diverse network environments as shown in table 5.2.

5.12. Success Rate and Reliability

Probe_success is one definite gauge of system reliability. With a value of 1, it assures that the probe was successful with no errors as shown in Fig 5.10. Such outcomes demonstrate the following:

- **Service Stability:** Requests in real time are processed smoothly by the infrastructure.
- **Readiness for Deployment:** The system is deployable to production-grade workloads.

Module	Target	Result	Debug
http_2xx	http://prometheus.io	Success	Logs
http_2xx	http://prometheus.io	Success	Logs
http_2xx	http://54.86.8.218:31388	Success	Logs
http_2xx	http://prometheus.io	Success	Logs
http_2xx	http://54.86.8.218:31388	Success	Logs
http_2xx	http://54.86.8.218:31388	**Failure**	Logs
http_2xx	http://prometheus.io	Success	Logs
http_2xx	http://prometheus.io	Success	Logs
http_2xx	http://54.86.8.218:31388	Success	Logs
http_2xx	http://prometheus.io	Success	Logs
http_2xx	http://54.86.8.218:31388	Success	Logs
http_2xx	http://54.86.8.218:31388	**Failure**	Logs
http_2xx	http://prometheus.io	Success	Logs
http_2xx	http://54.86.8.218:31388	**Failure**	Logs
http_2xx	http://prometheus.io	Success	Logs
http_2xx	http://prometheus.io	Success	Logs
http_2xx	http://54.86.8.218:31388	Success	Logs
http_2xx	http://prometheus.io	Success	Logs
http_2xx	http://54.86.8.218:31388	Success	Logs
http_2xx	http://prometheus.io	Success	Logs
http_2xx	http://54.86.8.218:31388	Success	Logs

Fig. 5.10. Probes

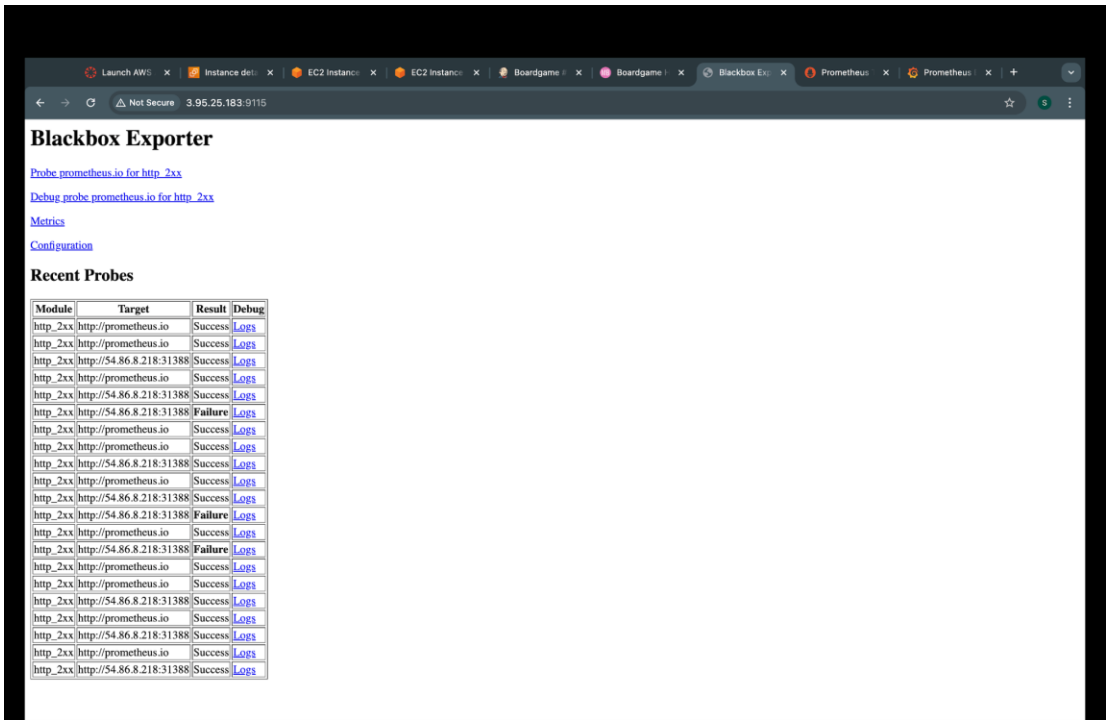

Fig. 5.11. Blackbox Application Dashboard of Logs

5.13. CI/CD Pipeline Validation

Here are the pipeline logs with different automated processes designed in the CI/CD pipeline:

5.13.1. Build and Deployment Stages:

There are strict, sequential build and deployment stages from compilation (MVN compile) to Docker image creation and Kubernetes deployment.

5.13.2. Static Code and Vulnerability Analysis:

The following tools have been used: Trivy and SonarQube for the pipeline to adhere to standards of security and quality. These tools check vulnerabilities in the codebase and images of containers, providing an active approach towards system hardening as shown in 5.12.

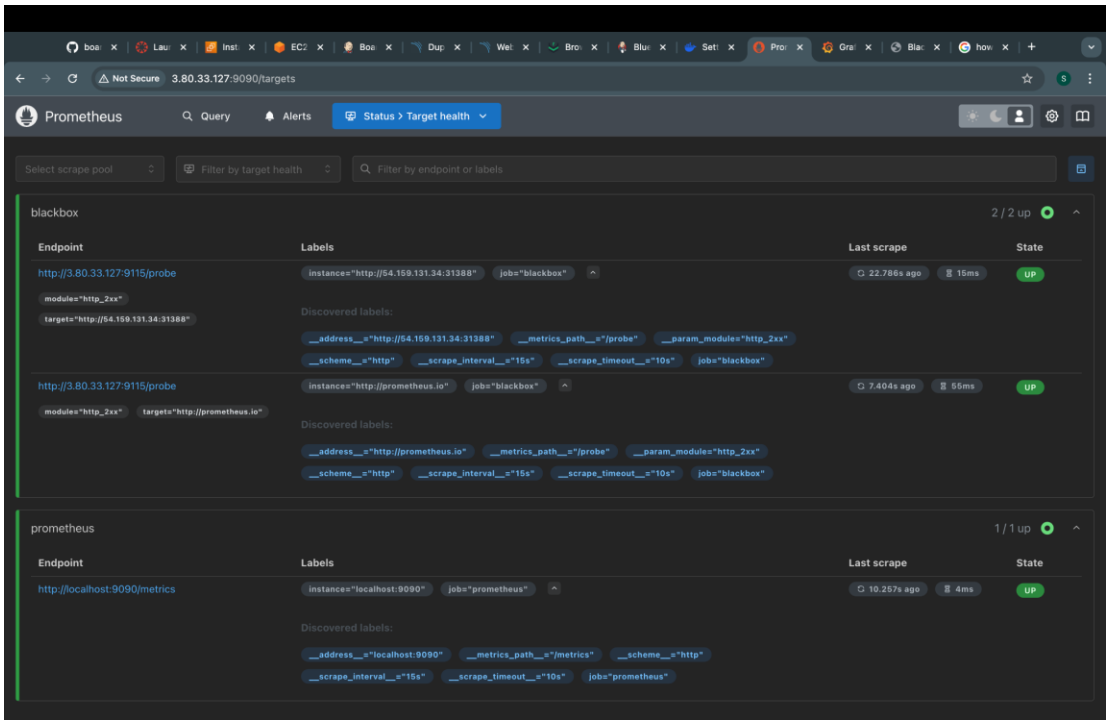

Fig. 5.12. Promethus Dashboard After Connection to the Application

5.13.3. Artifact Management:

Publish artifacts of build to Nexus, ensuring software components are centrally managed with a versioning system.

5.13.4. Notification Mechanisms:

Configured email notifications provide quick feedback loops, keeping interested parties abreast of pipeline health and performance.

Table 5.2. HTTP duration across phases provides granular insights

Phase	Duration (seconds)	Implication
resolve	0.000012185	Highly efficient DNS resolution.
connect	0.001711466	Quick connection setup, indicating optimal network conditions.
processing	0.007291979	Fast request processing, reflecting well-configured backend applications.
transfer	0.00057556	Minimal data transfer time, enabled by streamlined payloads and fast networks.
tls	0	No TLS handshake detected, highlighting a lack of secure communication.

CHAPTER 6

CONCLUSION AND FUTURE ENHANCEMENT

This project would provide an in-depth analysis of the performance of AWS cloud infrastructure, including its capability to host a Continuously Integrated and Deployed pipeline. This demonstrates the efficiency of AWS in managing workloads across different layers of compute, network, and applications in handling, at the same time, the challenging demands of an automated CI/CD pipeline. This work underlines the reliability of AWS in enabling modern DevOps practices by closely examining and evaluating the performance of key metrics and core components.

The construction of the CI/CD pipeline utilized AWS services to automate code integration, testing, and deployment processes. All stages of the pipeline-from source code management and build processes through deployment and monitoring-were carefully designed to ensure seamless transitions with minimum manual intervention. By incorporating secure testing checkpoints and infrastructure monitoring into the pipeline, the pipeline increased the speed of deployments while being robust and resilient.

The pipeline works well as of the adaptability of AWS services, including EC2 for compute, and monitoring tools such as Grafana. Continuous deployment proved to be seamless due to rapid iterations of application updates, with robust infrastructure performance even with fluctuating workloads. This approach minimizes time-to-market for applications while achieving stability and scalability.

The CPU utilization metrics depicted in the screenshots indicate the AWS infrastructure manages its compute resources well in the CI/CD pipeline. Low baseline CPU usage with short-lived spikes indicates that the system is able to function with periodic build and deployment workloads without stability implications. The dynamic scaling of the pipeline based on demand further underlines the suitability of AWS for resource-intensive DevOps workflows.

The network analysis demonstrated effective data transfer handling, which is crucial for functionalities such as artifact storage, uploading container images, and deployment updates within the CI/CD pipeline. The baseline performance of constant network output was complemented by efficient traffic spike handling to ensure continuous flow of data, irrespective of demand spikes. This reliability is pivotal in the seamless performance of pipeline stages, most commonly witnessed during the deployment activity across different regions or microservices.

Grafana's probe and HTTP metrics gave insight in real-time on what was happening operationally in the pipeline. Monitoring tools then allowed exact tracking of deployment time, resource consumption, and system latency to proactively identify and debug any issues. The periodic spikes during HTTP processing times indicate the necessity of moving performance monitoring more deeply into the pipeline, thus eliminating potential bottlenecks around potentially problematic build or deployment phases.

Building the CI/CD pipeline involved some challenges, including uniform workload distribution, minimal latency for deployments, and an integration of checkpoints for secure testing. These challenges were approached with intelligent scheduling of workloads, optimized allocation of resources, and robust automation frameworks. Containerized deployments using Docker and Kubernetes were incorporated to achieve improved efficiency and reliability in the pipeline.

Thus, this CI/CD pipeline showcases the power of transformation of DevOps methodologies for modern software engineering. Through automating routine work, maintaining consistency in deployment, and providing continuous feedback, the pipeline accelerates development cycles and brings development teams closer together. The paper serves as a practical guide for organizations seeking to implement DevOps practices in an AWS environment-the design of architecture, the optimization of resources, and the monitoring of performance.

6.1 Future work

The implementation of the CI/CD pipeline and performance analysis in this work demonstrate the capability of AWS infrastructure to handle modern workflows in DevOps. Still, as technology changes with more advanced methods of software development, there is much scope for future improvement and innovation. This section discusses some of the pathways for further research and development, including the scalability, robustness, and intelligence of the pipeline in order to address dynamic and complex software environments.

One of the promising areas for future work is integration with AI and ML technologies into the CI/CD pipeline. Predictive analytics can then be applied in forecasting performance bottlenecks, resource constraints, or other failures during deployment. Insights can be derived from historical pipeline data on patterns that lead to failures or inefficiency, thereby enabling proactive mitigation action. More significantly, AI-driven automation could dynamically adjust resource allocation during peak periods, optimizing both the performance and cost efficiency of the pipeline.

Although this research was narrowly focused on AWS, the organizations are increasing their adoption of multi-cloud or hybrid cloud strategies as they spread out their infrastructure and shun dependency on a single vendor. Future work could be conducted to extend CI/CD pipelines such that it works seamlessly across multiple platforms like Microsoft Azure and Google Cloud Platform, while maintaining consistency in performance. This would involve mechanisms for cross-platform orchestration, unified monitoring tools, and secured data transfer protocols between environments.

Future improvements could be in real-time vulnerability scanning to be embedded in the CI/CD pipeline and utilize tools like OWASP ZAP or Burp Suite for the dynamic testing of security and, more specifically, adopted frameworks with zero-trust security mechanisms. Finally, automating regulations during pipeline executions such as GDPR and HIPAA at their respective occurrences enables clean adherence to industry standards without the need for manual intervention.

While this study utilized Grafana for monitoring and insights, the growing complexity of software systems necessitates even more granular observability. Future pipelines could incorporate distributed tracing tools, such as Jaeger or Zipkin, to gain deeper visibility into microservices interactions and identify latency issues at the service level. Advanced dashboards with AI-driven anomaly detection could provide real-time alerts for unusual patterns in resource consumption, network traffic, or deployment metrics, ensuring faster incident resolution.

The future of CI/CD pipelines lies in the ability to adapt to emerging trends, integrate advanced technologies and escalate deployment complexity. Addressed areas will get better pipelines toward intelligence, security, and scalability as they continue to drive innovation in the software development arena. This shall provide a continuous evolution that keeps organizations ahead of the curve in delivering applications at high quality with minimal speed and efficiency.

REFERENCES

[1] Grande, Ruben, Aurora Vizcaino, and Felix O.Garica. “Is it worth adopting Devops practices in Global Software Engineering? Possible challenges and benefits.” Computer Standards & Interface 87(2024): 103767.

[2] Doan, Thien-Phuc, and Souhwan Jung. “DAVS: Dockerfile Analysis for Container Image Vulnerability Scanning.” Computer, Material & Continua 72.1(2022).

[3] Shan, Chenggang, et al. “KubeAdaptor: a docking framework for workflow containerization on Kubernetes.” Future Generation Computer System 148 (2023): 584-599.

[4] Lange,Moritz, et al. “Modern Build Automation for an Insurance Company Tool Selection.” Procedia Computer Science 219 (2023): 736-743

[5] Rubert, Maluane, and Leinner Farias. “On the effects of continuous delivery on code quality: A case study in industry.” Computer Standards & Interfaces 81 (2022): 103588.

[6] Wong, Ann Yi, et al. “On the security of containers: Threat modeling, attack analysis,and mitigation strategies.” Computer & Security 128 (2023): 103140

[7] Zhao, Zhiming, et al. “Developing and operating time critical applications in clouds: the state of the art and the SWITCH approach.” Procedia Computer Science 68 (2015): 17-28. [8] Jenkins 111. Arthur L., et al. “Redefining the Treatment of Lumbosacral Transitional Vertebrae for Bertolotti Syndrome: Long-Term Outcomes Utilizing The Jenkins Classification to Determine Treatment.” World Neurosurgery 175 (2023): e21-e29.

[9] Mills, Alan, Jonathan White, and Phole Legg. “Longitudinal risk-based security assessment of docker software container images.” Computers & Security 135 (2023): 103478.

[10] Steidl, Monika, Michael Felderer, and Rudolf Ramler. "The pipeline for the continuous development of artificial intelligence models—Current state of project and practice." Journal of Systems and Software 199 (2023): 111615.

[11] Eramo, Romina, et al. “An architecture for model-based and intelligent automation in DevOps.” Journal of Systems and Software 199(2023): 111615.

[12] ArulKumar,v, and R Lathamanju. “Start to finish automation achieve on cloud with build channel:By DevOps method.”Procedia Computer Science 165(2019):399-405.

[13] Amaro, Ricardo, Ruben pereria, and Miguel Mira da Silva. “Mapping DevOps capabilities to the software life cycle: A Systematic literature review.” Information and Software Technology(2024):107583.

[14] Kumar, Ankur, Mohammad Nadeem, and Mohammad Shameem.”Metaheuristic-based cost-effective predictive modelling for DevOps project success.”Applied Soft Computing(2024):111834.

[15] Cuadra, Julen, et al. “Enabling DevOps for Fog Applications in the Smart Manufacturing domain: A Model-Driven based Platform Engineering approach.” Future Generation Computer Systems 157 (2024): 360-375.

[16] Wiedemann, Anna, et al. “Integrating development and operations teams: A control approach for DevOps.” Information and Organization 33.3 (2023): 100474.

[17] Mishra, Alok, and Ziadoon Otaiwi. “DevOps and software quality: A systematic mapping.” Computer Science Review 38 (2020): 100308.

[18] Kovacs, Jozsef, Peter Kacsuk, and Mark Emodi. “Deploying docker Swarm cluster on hybrid clouds using octopus.” Advances in Engineering Software 125 (2018): 136-145.

[19] Penchalaiah, N. et al. “Clustered Single-Board Devices with Docker Container Big Stream Processing Architecture.” Computers, Materials & Continua 73.3 (2022).

[20] Guerreiro, Guilherme, et al. “A self-adapted swarm architecture to handle big data for “factories of the future”.” IFAC-reportsOnLine 52.13 (2019): 916-921.
[21] Kadri, Sabah, et al. "Containers in bioinformatics: applications, practical considerations, and best practices in molecular pathology.” The Journal of molecular diagnostics 24.5 (2022): 442-454.
[22] Yang, Dawei, et al. “DevOps in practice for education management information system at ECNU.” Procedia Computer Science 176 (2020): 1382-1391.

Vedhavathy T.R

mtech

Paper 4

NWC Class

SRM Institute of Science & Technology

Document Details

Submission ID

trn:oid:::1:3106859567

Submission Date

Dec 8, 2024, 10:52 PM GMT+5:30

Download Date

Dec 8, 2024, 10:55 PM GMT+5:30

File Name

for_plag_check.pdf

File Size

3.7 MB

53 Pages

11,221 Words

64,230 Characters

4% Overall Similarity

The combined total of all matches, including overlapping sources, for each database.

Filtered from the Report

- Bibliography
- Quoted Text

Match Groups

32 Not Cited or Quoted 4%
Matches with neither in-text citation nor quotation marks

0 Missing Quotations 0%
Matches that are still very similar to source material

0 Missing Citation 0%
Matches that have quotation marks, but no in-text citation

0 Cited and Quoted 0%
Matches with in-text citation present, but no quotation marks

Top Sources

3% Internet sources
2% Publications
2% Submitted works (Student Papers)

Integrity Flags

0 Integrity Flags for Review

No suspicious text manipulations found.

Our system's algorithms look deeply at a document for any inconsistencies that would set it apart from a normal submission. If we notice something strange, we flag it for you to review.

A Flag is not necessarily an indicator of a problem. However, we'd recommend you focus your attention there for further review.

Match Groups

- **32** Not Cited or Quoted 4%
  Matches with neither in-text citation nor quotation marks
- **0** Missing Quotations 0%
  Matches that are still very similar to source material
- **0** Missing Citation 0%
  Matches that have quotation marks, but no in-text citation
- **0** Cited and Quoted 0%
  Matches with in-text citation present, but no quotation marks

Top Sources

3% Internet sources
2% Publications
2% Submitted works (Student Papers)

Top Sources

The sources with the highest number of matches within the submission. Overlapping sources will not be displayed.

1	Publication	Igwe, Hephzibah. "The Significance of Automating the Integration of Security and... 1%
2	Internet	www.coursehero.com 1%
3	Internet	dev.to 0%
4	Internet	docs.caascad.com 0%
5	Student papers	Asia Pacific University College of Technology and Innovation (UCTI) 0%
6	Student papers	University of Carthage 0%
7	Student papers	Leeds Beckett University 0%
8	Publication	Nick Alteen, Jennifer Fisher, Casey Gerena, Wes Gruver, Asim Jalis, Heiwad Osman... 0%
9	Internet	www.geeksforgeeks.org 0%
10	Publication	Nikhil Pathania. "Pro Continuous Delivery", Springer Science and Business Media ... 0%