Trung Hoang Nguyen

A Comprehensive CI/CD Pipeline and Google Cloud Platform Deployment for Web Application

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communication Technology

Bachelor’s Thesis

10 October 2023

Abstract

Author:	Trung Hoang Nguyen
Title:	A Comprehensive CI/CD Pipeline and Google Cloud Deployment for web application
Number of Pages:	46 pages
Date:	10 October 2021
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communication Technology
Professional Major:	Smart IoT Systems
Supervisors:	Tapio Wikström, Senior Lecturer

The goal of this project was to explore the term "DevOps" culture in software development and implement a Continuous Integration, Continuous Delivery/Development (CI/CD) pipeline for a web application. The thesis is divided into two main parts: the theoretical background and the project section. The first section will explain the principles of DevOps and its importance in software development. It will also study and analyze the technology stacks related to DevOps. Additionally, this part discusses the tool selection that was used in the project.

The project section focuses on the methodology used to implement the CI/CD pipeline and an overview of the React web application. The pipeline can be initiated by pushing code into to the repository and triggering the CI tool. The code is pulled from the code source management and analyzed. If successful, the containerization tool packages the code as a container and sends it to the container registry. Google Cloud then provides the cloud infrastructure, which is managed by Terraform, an infrastructure as a code tool. Two cloud-based instances serve as the testing and production environments, pulling the container and starting it. A monitoring solution will be deployed to observe the project remotely.

One of the outcomes includes the successful deployment of the pipeline, which consistently delivers software to both environments within three minutes. The second outcome is the implementation of containers in software development. Moreover, the project also successfully applies Google Cloud and Terraform for cloud infrastructure management, demonstrating the benefits of Infrastructure as Code (IaC).

In conclusion, the project achieved reusability through modular design, making it adaptable for future projects. It acts as a foundation for future CI/CD work and potential exploration in DevOps and Cloud Computing.

Keywords: DevOps, containerization, continuous integration, continuous delivery, continuous deployment, containerization, infrastructure as code, cloud computing, Jenkins, Docker, Terraform, Google Cloud Platform, GitHub, SonarQube

Contents

List of Abbreviations

1 Introduction 1

2 Theoretical Background 2

2.1 Overview of DevOps 2
2.2 Explanation of CI/CD Concept 3
2.2.1 What Is CI? 3
2.2.2 What Is CD? 4
2.3 The Benefits of Implementing CI/CD in Software Development 4

3 Project Methodology and Tools 6

3.1 Tools and Technologies Selection 6
3.1.1 Jenkins 6
3.1.2 Docker 7
3.1.3 SonarQube 9
3.1.4 Prometheus 9
3.1.5 Grafana 10
3.1.6 Node Exporter 11
3.1.7 Terraform 11
3.1.8 Google Cloud Platform 12
3.2 Project Method and Architecture 13
3.2.1 Cloud Infrastructure 13
3.2.2 CI/CD Pipeline 14
3.2.3 MooMoo Web Application 15

4 Implementation 19

4.1 Infrastructure as Code 19
4.2 GitHub 21
4.3 Docker and Dockerfile 23
4.4 SonarQube 25
4.5 Jenkins 27
4.6 Configuring CI/CD Pipeline 30
4.7 Monitoring Solution 34

5 Results and Challenges 39

5.1 Project Results 39
5.2 Project Challenges 40
5.3 Further Improvements 41

6 Conclusion 42

Bibliography 44

List of Abbreviations

API	Application Programming Interface
CD	Continuous Development or Continuous Deployment
CI	Continuous Integration
Dev	Development
DOM	Document Object Model
GCP	Google Cloud Platform
HTTP	Hypertext Transfer Protocol
IaC	Infrastructure as Code
IT	Information Technology
JDK	Java Development Kit
Npm	Node Package Manager
Ops	Operations
OS	Operating System
SSH	Secure Socket Shell
UI	User Interface
URL	Uniform Resource Locator
VM	Virtual Machine

1 Introduction

The last decade has witnessed an increasing demand for digital transformation among all sectors. For all businesses, software and web applications have become the key to success. However, the traditional approach to software development has become outdated. The reason is that releases occur relatively infrequently, such as once every six months. During this time, all bug fixes and new features are accumulated and then bundled into one large update, resulting in increased frequency of human errors. Moreover, the development team might suffer from constant pressure from the management to shorten the time-to-market as much as possible. This challenge was resolved with the emergence of a revolutionary concept – DevOps.

The ultimate goal of DevOps is to consistently deliver efficient and reliable applications to consumers. One of the key components of DevOps is Continuous Integration and Continuous Development (CI/CD), which enables automated and frequent code integration, testing, and deployment [1]. By adopting CI/CD practices, companies can accelerate software delivery, reduce errors, and ensure the continuous improvement of their applications. This final year project resolves around DevOps, with a particular emphasis on demonstrating CI/CD pipelines for a website.

This is divided into five main sections. In Section 2 (**Theoretical Background**), an explanation of essential concepts related to DevOps and CI/CD will be provided. Next, the third section (**Project Methodology and Tools**) focuses on the tools and technologies available for building CI/CD pipelines. Then, the implementation plan will be further analysed in Section 4 (**Implementation).** The Section 5 (**Results and Challenges**) presents outcomes and practical implications of the study. Finally, an overall conclusion will be drawn in the Section 6 (**Conclusion**).

2 Theoretical Background

2.1 Overview of DevOps

DevOps is a combination of two terms which are software development (Dev) and operations (Ops). This notion was first used by John Allspaw and Paul Hammond during a presentation at Flickr in 2009 [2]. In simple words, DevOps is an agile approach that delivers applications and services at scale by bring together software developers and operations teams [3]. Under this framework, the role of each player is well defined.

- A security team ensures a safe delivery of software.
- An engineer team is responsible for planning, developing, and verifying code.
- An operation teams ensures successful release, configuration, and monitoring.

This can also be seen as an organizational culture shift that emphasized transparency, communication, and collaboration between teams. Shared responsibilities are the essence of this culture when different teams within an organization are responsible for the success or failure of a product. The engineer teams will now understand more about the challenges faced by operations; hence they are more likely to simplify deployment. In contrast, operations teams can contribute by providing fast feedback and adopting automation tools [4].

Nurturing the DevOps culture within organizations requires integrating various tools such as pipelines, code mergers, and other automation techniques in order to quickly move code from design to deployment [5 pp. 210]. This is directly related to Continuous Integration / Continuous Delivery (CI/CD) – A fundamental instrument of DevOps. In the next sections, this instrument is going to be further analysed.

2.2 Explanation of CI/CD Concept

As mentioned earlier, CI/CD is a practical implementation of the principles of DevOps combined by Continuous Integration (CI) and Continuous Delivery or Continuous Development (CD). Figure 1 illustrates the workflow of a CI/CD pipeline.

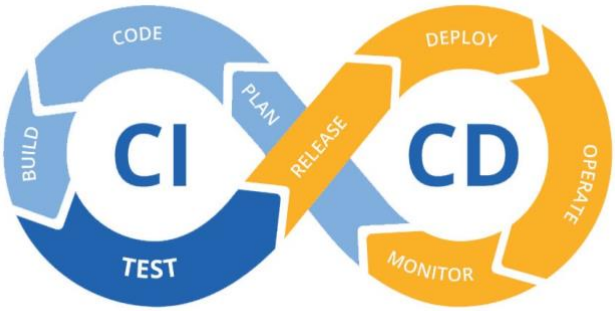

Figure 1. Example of CI/CD pipeline [6].

In the above figure, CI and CD are strongly connected, creating a step-by-step approach to continuously code, build, test, release, and monitor a software product.

2.2.1 What Is CI?

The 'CI' refers to an automation process that executes a pre-defined automated build script at each stage. Whenever there are new code changes, the CI process will be triggered to build, test, and merge the changes to the repository [7]. By implementing CI, the codebase can be more stable, any issues or conflicts are quickly identified and resolved continuously. Moreover, this not only improves the overall efficiency of development process but also enhances the quality and reliability.

2.2.2 What Is CD?

“CD” in CI/CD can be related to two terms - Continuous Delivery or Continuous Deployment, which is carried out after the CI process to make sure that changed code can be delivered to customers in an error-free way. The two terms have similar concept about automating further stages of the pipeline but in some cases, they are used separately to demonstrate how automation is happening [7].

On the one hand, CD allows changes to be automatically checked and uploaded to code repository. This ensures that the latest version of the code is always available for further development or testing. On the other hand, CD focuses on streamlining the process of delivering code changes to a staging environment, where further testing and validation can take place. It allows for a more frequent and reliable release cycle, enabling development teams to iterate and improve their software more rapidly.

Besides, CD is the final stage of CI/CD pipeline. It involves automatically deploying code from repository to production environment, making it available to end-users. This eliminates the need for manual intervention in the deployment process and allows a continuous flow of changes from development to production. With the integration of CI/CD pipeline, developers can ensure a faster time-to-market and reduce the risk of human errors during the deployment process [8].

2.3 The Benefits of Implementing CI/CD in Software Development

Integrating CI/CD to software development projects can provide many benefits for businesses. In general, it promotes automation processes, stability, and rapid development of software updates [9]. All detailed advantages can be summarized in Table 1.

1	**Rapid feedback loop**	With CI/CD, immediate feedback will resolve all information silos between teams. For instance, a customer reports that the web application has an interface bug, and he/she cannot fill out the contact form. Developers will be notified immediately to update the code. Once the testing is successfully finished, the code will automatically run.
2	**Real-time updates**	The goal of every web application is to stay up to date and meet the needs of users. CI/CD allows innovative features to be integrated in the app in real-time thanks to automated pipeline.
3	**Quick fault isolation**	Instead of huge chunks of code, developers now release code in small pieces in a shared repository. They collaborate to find bugs, ensuring that no error can make it to production. Thus, CI/CD can promote high-quality releases with fewer bugs.
4	**Increased test reliability**	CI/CD allows making small code changes. These bite-size pieces of code can be tested faster as soon as they are saved in the repository. Hence, a larger amount of accurate positive and negative tests can be proceeded. This is also called Continuous Reliability. One common metric to measure reliability is Mean Time to Resolution (MTTR). In addition CI/CD helps developers to reduce this number by small code changes and quick tests [10].
5	**Collaboration and Communication**	CI/CD improves accountability and communication between developers and operational teams who work on a similar project. Moreover, stakeholders can be notified about project status, changes, or even failure.

Table 1: Advantages of CI/CD.

3 Project Methodology and Tools

The key points related to methodology will be divided into two sub-sections. Firstly, an in-depth introduction to important tools and cloud service is given. The selection of these tools depended on several aspects including the specific requirements and scope of this project. Then, the structure of CI/CD pipeline and used cloud services are extensively explored in the second sub-section.

3.1 Tools and Technologies Selection

3.1.1 Jenkins

There are several CI tools but during this project only utilized and compared some popular ones such as Jenkins, TravisCI and GitHub Action. When choosing the right CI software for this project, several factors were considered such as the versatility of plugins, compatibility, and integrations or support for Docker container. A thorough comparison of these tools is depicted in the Table 2.

Tools	TravisCI [11]	GitHub Actions [12]	Jenkins [13]
Open source	No	No	Yes
Supporting platforms	Linux MacOS Windows Docker (except on MacOS)	Linux Windows MacOS	Linux Windows MacOS Docker Kubernetes
Containerization	Yes	Yes	Yes
Hosting	On-premised Cloud	Cloud	On-premised
Free version	Limit on existing job	Free for public repositories Paid for private	Yes
Advantages	Fast start	GitHub ecosystem	Limitless Plugins Incredible support

	Lightweight YAML config No dedicated server-required		Free Automation Integrated
Disadvantages	Limited plugins, Lack of scalability Price	Limited features	Poor UI

Table 2: Comparison of CI tools.

After consideration, the author decided to use Jenkins in this project based on its superiority in features and large community support.

3.1.2 Docker

Containers are lightweight packages of applications with necessary elements to run in any environment by virtualizing an operating system. Compared to virtual machines (VMs), containers do not include the entire OS. They only contain the necessary OS processes and dependencies to execute code [14]. This makes the container's size is smaller which is measured in megabytes (compared to gigabytes for VMs), allowing hardware to run faster and simplifying the process of moving the application between different environments (such as development, testing, and production) while maintaining its full functionality. As a result, this technology offers incredible benefits including isolation, cost-effective scalability, and improved developer productivity [14].

Docker stands as an open-source and highly favored tool for containerization. It provides access to native containerization capabilities through user-friendly interface and also offers several advantages [15]:

- Simplicity and faster configurations
- Compatibility, maintainability and portability
- Resource efficiency
- Reusable

- CI efficiency
- Extensive community support

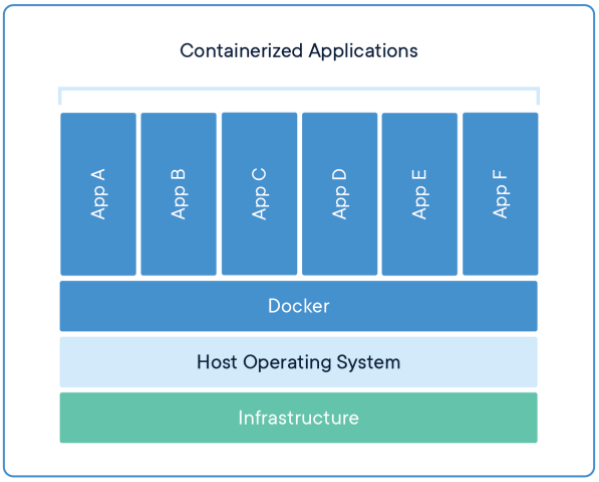

Figure 2. Demo of containerization [15].

The Docker architecture operates on a client-server model, where the Docker client interacts with the Docker daemon. The daemon is responsible for tasks like building, running, and distributing Docker containers. The client and daemon can either coexist on a single system or connect remotely, with communication facilitated through a REST API over UNIX sockets or a network interface. Docker Compose is an additional client that enables working with multi-container-based applications. The figure below demonstrates the inner mechanism of Docker.

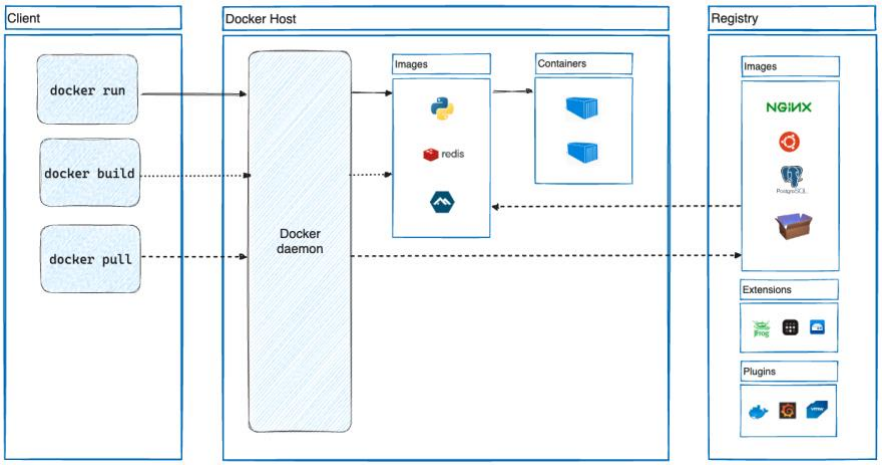

Figure 3. Docker Architecture [15].

3.1.3 SonarQube

SonarQube is a self-managed Code Quality Assurance tool that provides continuous scanning, analysis, and evaluation of code quality. It can generate metrics and statistics at each level to identify areas in the source code that require improvement [16].

One of the crucial features of the SonarQube scanning progress is Quality Gate which is a checkpoint in the software development process that must be passed before the code can move forward. It is a way to ensure that the code meets certain quality standards, such as being clean, well-tested, and secure. Quality Gates cover several aspects such as [16]:

- Code style: Does the code follow the required code's style guide?
- Security vulnerability: Are there any threads in the code?
- Code complexity: How complex is the code?
- Code coverage: How is the code's result?

If the code passes all quality checks, the Quality Gate is considered to be passing (green). This means the code is ready to be merged into the main codebase or deployed to production. If any check fails, the Quality Gate returns a failing status (red), indicating that the code needs to be re-evaluated and further action should be taken. SonarQube provides in-depth guidance for each issue and offers suitable solutions to effectively address them.

3.1.4 Prometheus

Prometheus is an open-source monitoring solution that is based on time series databases. It collects metrics along with a unique ID and timestamp [17]. Metrics can help developers to dive inside their applications and give them magical numbers to understand what make their application run more slowly and deal with it. Figure 4 illustrates how the operation of Prometheus works.

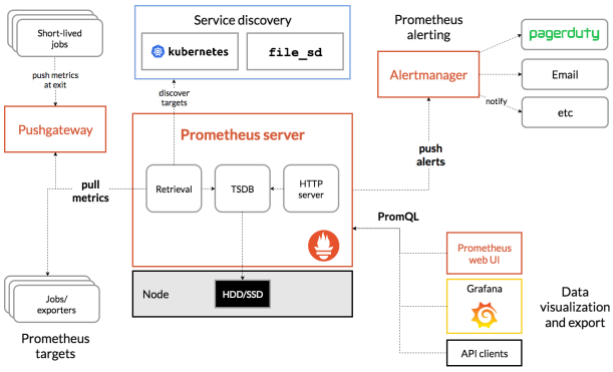

Figure 4. Prometheus Architecture [17].

3.1.5 Grafana

During this project, a large amount of data needs to be analyzed. Grafana, an open-source solution for data analysis and visualization, was used to assist with this task [18]. It allows developers to easily create appealing dashboards to visualize metrics and gain insights into the performance and status of the application. Additionally, Grafana is a versatile monitoring and visualization tool that can be combined with various data sources, time series databases, or log data.

Furthermore, Grafana helps users analyze and monitor data over a period of time, known as time series analytics. In this project, Grafana was used to track memory usage, CPU usage, and CI/CD pipeline performance. Figure 5 shows an example of the monitoring process for the Linux system using Grafana.

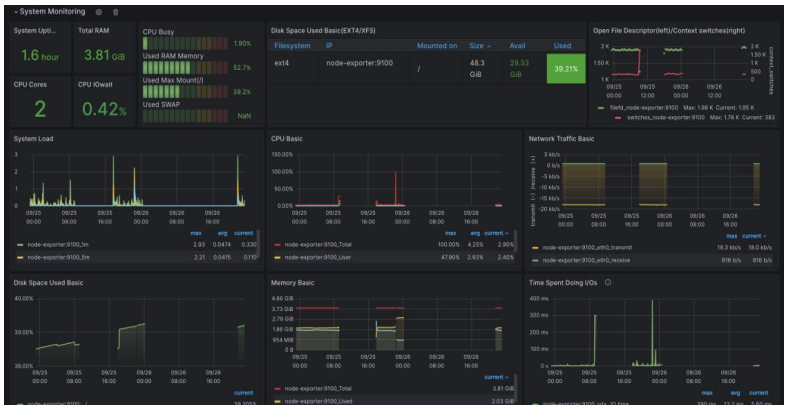

Figure 5. Monitoring Linux system using Grafana [18].

3.1.6 Node Exporter

Node Exporter is a highly efficient and powerful open-source Prometheus exporter tool [19]. It provides the capability to effortlessly monitor and gather a wide range of hardware and operating system metrics. These metrics include CPU usage, memory utilization, disk usage, and network traffic. With Node Exporter, it is easy to capture crucial system information and transmit it to Prometheus, which acts as a centralized hub for data aggregation and storage. Once the data is collected, users can leverage the visualization capabilities of Grafana to gain valuable insights and analyze the performance of their system. Installing Node Exporter is simple, as it can be obtained from the official Prometheus website or run as a Docker container with port 9100 exposed.

3.1.7 Terraform

Terraform represents an open-source Infrastructure as Code (IaC) solution created by HashiCorp [20]. It enables developers to define cloud and on-premises resources using easy-to-read, versionable, reusable, and shareable configuration files. It provides a consistent workflow for provisioning and managing all aspects of an infrastructure's lifecycle. Terraform has the capability to handle various types of components, including lower-level elements such as compute, storage, and networking resources, as well as higher-level components such as DNS entries and SaaS features. It accomplishes this by utilizing APIs to create and manage resources on cloud platforms and other services.

One of the key advantages of Terraform is its extensive feature set. It offers a wide range of functionalities that enable users to define and manage their infrastructure with ease from defining resources to creating complex environments [21]. Moreover, Terraform supports multiple cloud platforms, including AWS, Azure, and Google Cloud. This flexibility allows users to seamlessly work with their preferred cloud providers and take advantage of the unique features and services offered by each platform.

3.1.8 Google Cloud Platform

Cloud computing is a rapidly growing field that involves the delivery of computing services over the internet [22]. Services vary from servers, storage, databases, analytics, to intelligence. In the cloud computing market, there are three principal players including Amazon Web Services, Azure, and Google Cloud. Out of these three providers, this project will solely focus on Google Cloud because of its utilities.

Google Cloud Platform (GCP) is a comprehensive and robust cloud computing platform provided by Google. It offers a wide range of cloud services for computing, data storage, and application development. GCP covers application, storage, and cloud computing services for backend, mobile, and web solutions over the internet. The services are categorized into four main areas: Compute Engine, Cloud Storage, Big Data, and Services [23].

In Q2 2023, Google Cloud's growth rate reached 31%, the strongest growth of the top three market leaders, to capture 9% of the cloud market [24]. The figure bellow summarizes the firm's constant revenue growth during a six-year period.

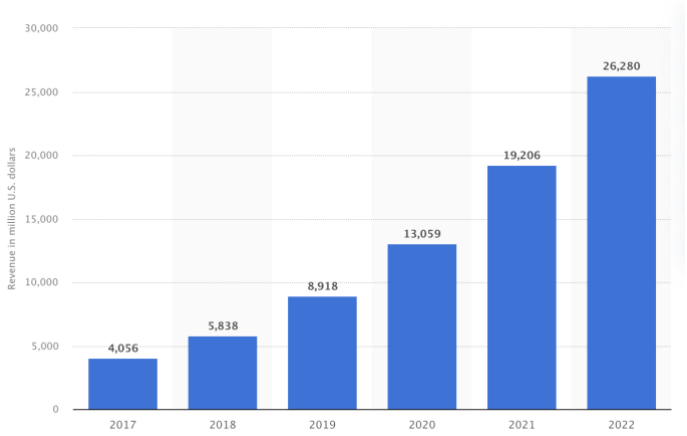

Figure 6. Global Google Cloud revenues from 2017 to 2022 (Statista) [25].

While there are competitors in the market offering similar functionalities, Google has been trying to differentiate itself by reducing costs for end consumers and adopting environmental-friendly practices by, for example:

- Designing energy-efficient servers and data centers by eliminating unnecessary components
- Incorporating built-in modules and adopting carbon-neutral practices.

3.2 Project Method and Architecture

After having explained the essential tools and technologies that were used in this project, the next sections focus on the Google Cloud and the CI/CD pipeline structure and give an overview of the movie web application.

3.2.1 Cloud Infrastructure

The deployment of project will be hosted on Google Cloud, by three virtual machines including:

- devops-app-vm: Production environment
- devops-jenkins-vm: Jenkins hosted, development environment and monitoring server.
- devops-sonarqube-vm: Sonarqube server

Figure 7 illustrates the GCP structure in which Terraform serves as orchestration engine.

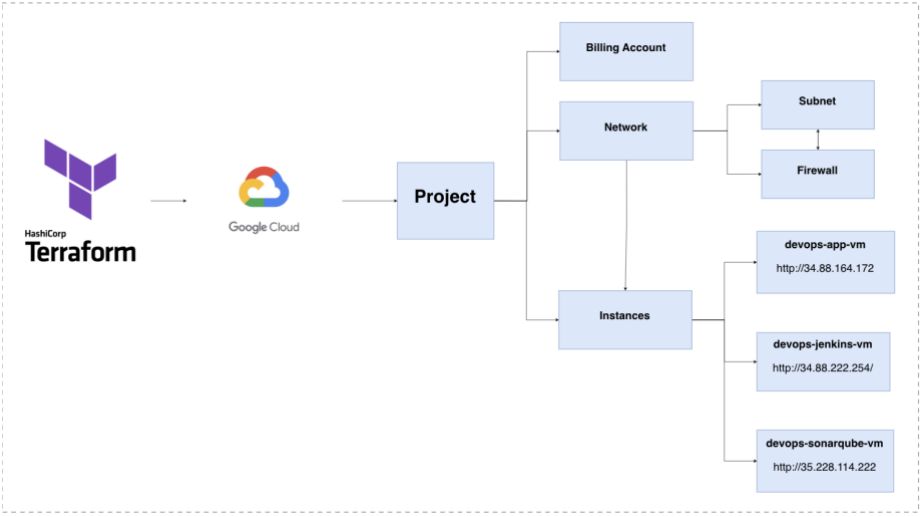

Figure 7. GCP architecture.

As illustrated in Figure 7, Terraform initiates the essential GCP services, ensures requisite APIs are enabled, creates a billing account for this project and establishes a network and a firewall to enable communication within instances and a connection from developers to cloud.

3.2.2 CI/CD Pipeline

As Figure 8 shows, the pipeline goes through several steps. It starts with the developer pushing code to Code Management (GitHub). Jenkins then receives the pushed command via webhook, triggering the job pipeline. Jenkins pulls the latest code from the repository and delivers it to SonarQube for quality checking on the “*devops-sonarqube-vm*” instance.

Once the code passes the quality check, Docker comes into play. It builds the code into an image, tags it, and sends it to the image registry. With Docker installed on Jenkins's instance, the code is deployed as a container. It is then ready for developer testing before Jenkins sends it to the production environment (devops-app-vm) via the SSH protocol. To monitor and visualize the pipeline, Prometheus and node exporters collect metrics from Jenkins and the hosted resources. These metrics are displayed on a dashboard through Grafana.

Finally, after completing all these stages, Jenkins sends a Telegram message to the developer, providing the status of the CI/CD pipeline.

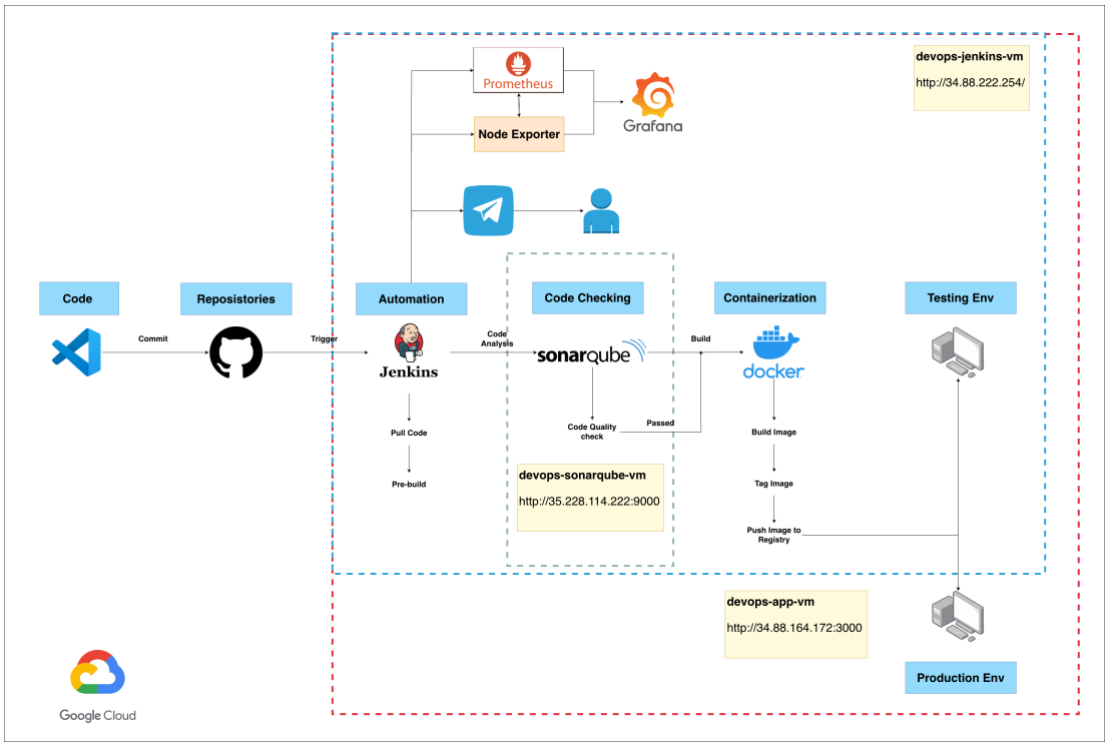

Figure 8. CI/CD pipeline.

3.2.3 MooMoo Web Application

The MooMoo web application is built upon a modularized architecture and is based on ReactJS - an efficient, declarative, and increasingly popular JavaScript library. ReactJS was created by Jordan Walke, a software engineer at Facebook. Initially developed and maintained by Facebook, ReactJS has been widely adopted and used in products such as WhatsApp and Instagram. Facebook started developing ReactJS in 2011 for its newsfeed section and released it to the public in May 2013 [26].

A ReactJS application consists of several components, where each component is responsible for generating a small, reusable section of HTML code. Components can be nested within other components, allowing for the construction of complex applications from simple building blocks. Moreover, ReactJS uses virtual DOM, which is a replica of DOM. When a web is modified, ReactJS updates the virtual DOM and compares it to the real DOM and then DOM updates the parts that have changed recently, leaving the rest unchanged.

This makes React minimizes the need for extensive updates to the actual DOM, resulting in enhanced performance and responsiveness [27].

```
class Car extends React.Component {
  render() {
    return <h2>Hi, I am a Car!</h2>;
  }
}
```

Figure 9. Example of React Component.

MooMoo is a website where entertainment meets convenience. The web application was inspired by Netflix - a top-tier movie streaming platform. With a user-friendly interface and a vast category of content, MooMoo provides a wide array of options to suit one's preferences and enhance movie selection experience.

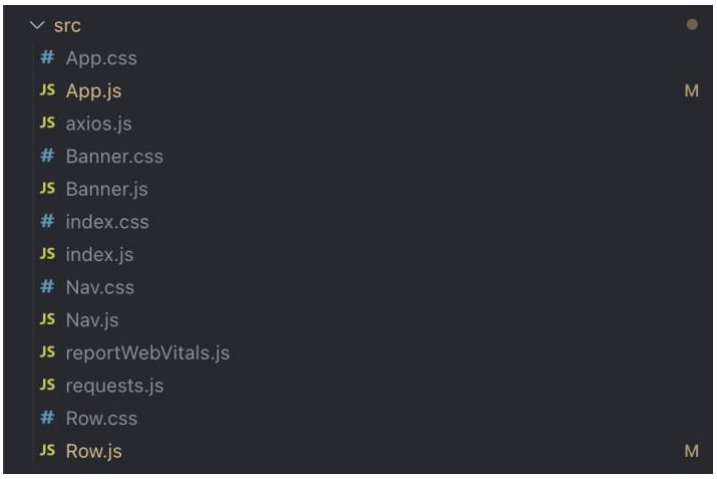

Figure 10. MooMoo code structure.

Figure 10 above demonstrates the core file structure. 'App.js' is rendered MooMoo which composes several modular components for reusability such as:

- Nav.js: Renders the top navigation bar, featuring logo and user avatar.
- Banner.js: Displays feature content, titles, and descriptions at the top of the page

- Row.js: Generates rows of a movie or TV shows trailer, allows user interaction.

Movie and TV shows trailers are retrieved from the API backend through Axios – a reliable and widely-used HTTP client. The "*requests.js*" is responsible for handling the API request. The categories are divided into multiple genres, allowing users to easily browse and discover trailers based on their preferences. (Figure 11)

```
const requests = {
    fetchTrending: `/trending/all/week?api_key=${API_KEY}&language=en-US`,
    fetchNetflixOriginals: `/discover/tv?api_key=${API_KEY}&with_networks=213`,
    fetchTopRated: `/movie/top_rated?api_key=${API_KEY}&language=en-US`,
    fetchActionMovies: `/discover/movie?api_key=${API_KEY}&with_genres=28`,
    fetchComedyMovies: `/discover/movie?api_key=${API_KEY}&with_genres=35`,
    fetchHorrorMovies: `/discover/movie?api_key=${API_KEY}&with_genres=27`,
    fetchRomanceMovies: `/discover/movie?api_key=${API_KEY}&with_genres=10749`,
    fetchDocumantaries: `/discover/movie?api_key=${API_KEY}&with_genres=99`
}

export default requests;
```

Figure 11. Extract of request.js file.

Once data is retrieved, it will be displayed to the user. A 'handle click' function will be responsible for managing the display of movie trailers when a movie poster is clicked.

```
const handleClick = (movie) => {
    if (trailerUrl) {
      setTrailerUrl('');
    } else {
      movieTrailer(movie?.name || movie?.original_title || "")
        .then(url => {
          console.log(url)
          const urlParams = new URLSearchParams(new URL(url).search)
          setTrailerUrl(urlParams.get('v'));
        })
        .catch(error => console.log(error));
    }
  }
```

Figure 12. Extract of Row.js file.

As can be seen, the function employs a conditional structure to determine if a video is currently being displayed. If the user clicks on it, the movie poster will close the video. On the other hand, if no trailer is currently played, the function

will call the "*movieTrailer*" library to fetch data. Figure 13 showcases the appearance of MooMoo running on the Docker container.

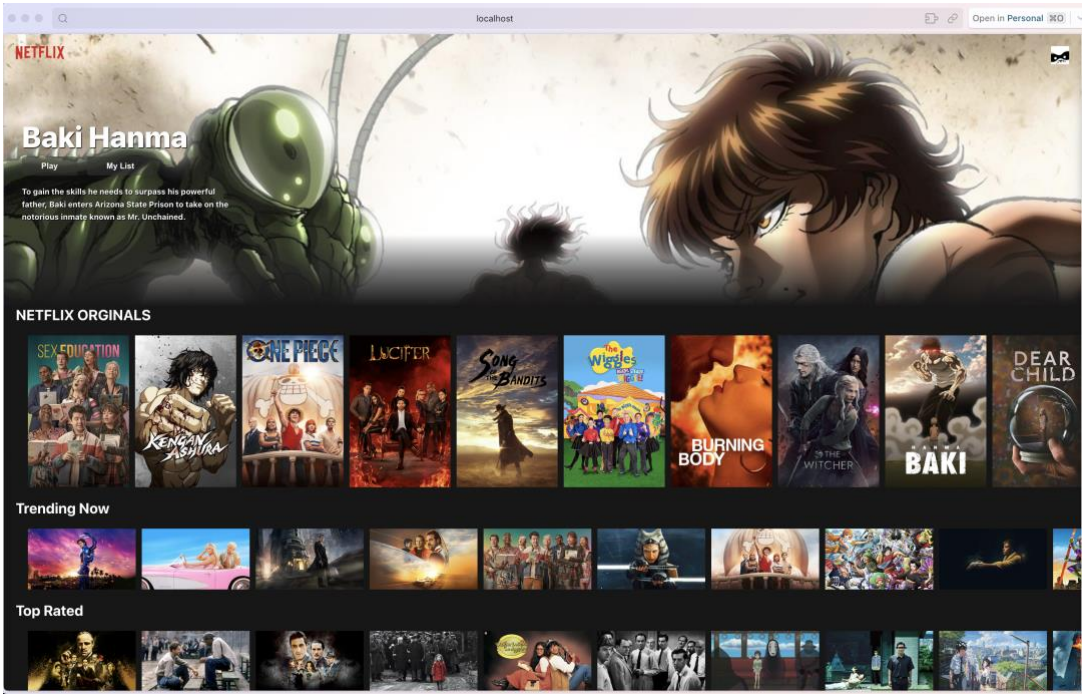

Figure 13. Demo of MooMoo on Docker container.

4 Implementation

4.1 Infrastructure as Code

As mentioned above, Terraform will be used to initialize infrastructure by Terraform configuration files. There are four crucial files:

- providers.tf: Starts Terraform on the Google Cloud terminal and enables the necessary API such as Compute Engine, Billing API or Cloud Storage API.
- main.tf: Creates virtual instances with the configured network, static IP, tags, and machine type.
- network.tf: Establishes the network within the project scope, configures the firewall which defines the network protocol and ports.
- static-ip.tf: Creates and assigns static IP addresses for each instance.

```
1   ##Create devops-app-vm instance
2   resource "google_compute_instance" "default" {
3     project                   = google_project.my_project.project_id
4     name                      = "devops-app-vm"
5     machine_type              = "e2-small"
6     zone                      = "europe-north1-a"
7     allow_stopping_for_update = true
8
9     boot_disk {
10      initialize_params {
11        image = "ubuntu-os-cloud/ubuntu-2204-lts"
12        size  = 50
13      }
14    }
15    network_interface {
16      subnetwork         = google_compute_network.network.name
17      subnetwork_project = google_project.my_project.project_id
18
19      access_config {
20        nat_ip = google_compute_address.dev_static_ip.address ##Assign static IP
21      }
22    }
23
24    tags = ["web"]
25
26    lifecycle {
27      ignore_changes = [
28        metadata
29      ]
30    }
31
```

Figure 14. Extract of main.tf to create configured devops-app-vm instance.

```
##Create new project
resource "google_project" "my_project" {
  name            = "devops-project"
  project_id      = "${random_string.project_prefix.result}-devops-project"
  billing_account = "014A84-EA1D72-E841F5"
}

##Enable Billing API
resource "google_project_service" "billingapi" {
  service = "cloudbilling.googleapis.com"
  project = google_project.my_project.id
}

##Enable Cloud Storage API
resource "google_project_service" "project" {
  project = google_project.my_project.id
  service = "storage.googleapis.com"

  timeouts {
    create = "30m"
    update = "40m"
  }

  disable_dependent_services = true
}

##Enable Compute Engine API
resource "google_project_service" "compute" {
  project = google_project.my_project.id
  service = "compute.googleapis.com"

  timeouts {
    create = "30m"
    update = "40m"
  }

  disable_dependent_services = true
}
```

Figure 15. Extract of provider.tf to create project and enable APIs.

After conducting research on the hardware requirements for each tool and considering the machine type configuration, the author has decided to utilize two different machine types for the instances. The SonarQube server and the application production server will be configured with the "*e2-small*" setup, which is a popular machine type that offers optimized cost and performance for general purposes such as lightweight websites and containerization. On the other hand, the Jenkins server will handle heavier workloads, such as running CI/CD pipelines, managing monitoring tools, and serving as a test environment. Therefore, it requires more hardware resources. The most suitable option is to use the "*e2-medium*" configuration, which offers faster performance than the small ones while still optimizing costs. All instances will install Linux Ubuntu, a highly efficient and light-weight operating system that is widely favored by developers and that has a strong support network within the community [28].

The first step to start Terraform is using the "*terraform init*" command. This will trigger terraform plugins integrated in Google Cloud, download necessary dependencies, and set up the work directory.

```
htrung_jobs@cloudshell:~ (devops-app-pipeline)$ terraform init

Initializing the backend...

Initializing provider plugins...
- Finding latest version of hashicorp/random...
- Finding hashicorp/google versions matching "~> 4.68.0"...
- Installing hashicorp/random v3.5.1...
- Installed hashicorp/random v3.5.1 (signed by HashiCorp)
- Installing hashicorp/google v4.68.0...
- Installed hashicorp/google v4.68.0 (signed by HashiCorp)

Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

Figure 16. Demo of using terraform init command.

After initializing Terraform, the subsequent step is to use the *"terraform plan*" command to review the upcoming creations or modifications before executing "*terraform apply*" to apply the infrastructure. After using Terraform, a new project is created with three instances. These instances are configured with static IP addresses.

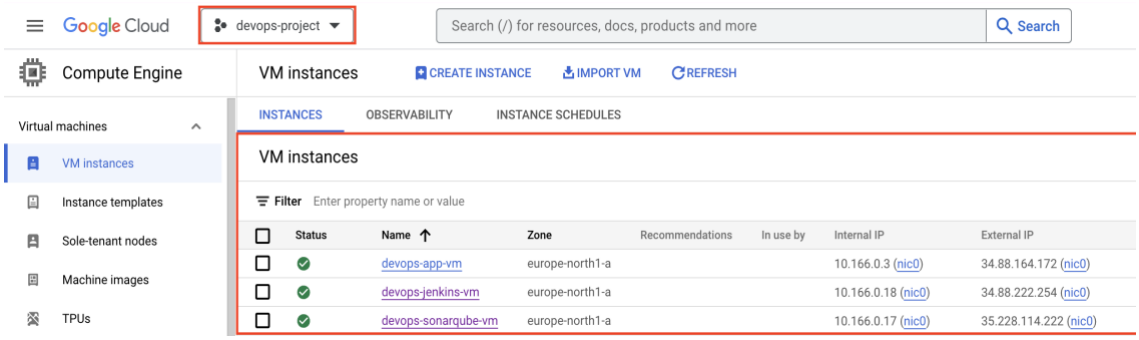

Figure 17. Outcome after applying Terraform.

4.2 GitHub

The management of codebase is orchestrated through GitHub, a popular version control tool. The purpose of project was to automate the CI/CD process, in order to achieve that, the webhook must be configured with Jenkins's IP addresses.

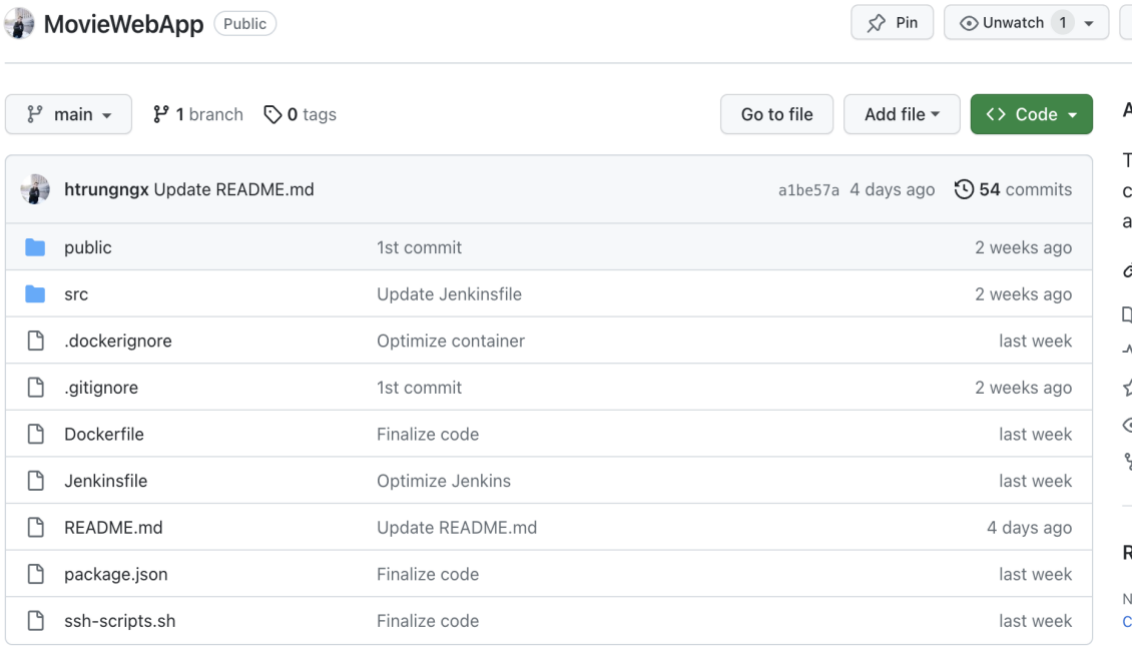

Figure 18. Code Repository.

To trigger the Jenkins job, the pipeline requires a webhook. This webhook serves as a mechanism for GitHub to notify Jenkins of any changes to the repository. The webhook is formatted using Jenkins's URL with "*/github-webhook/*" appended to it. (Figure 19) [29].

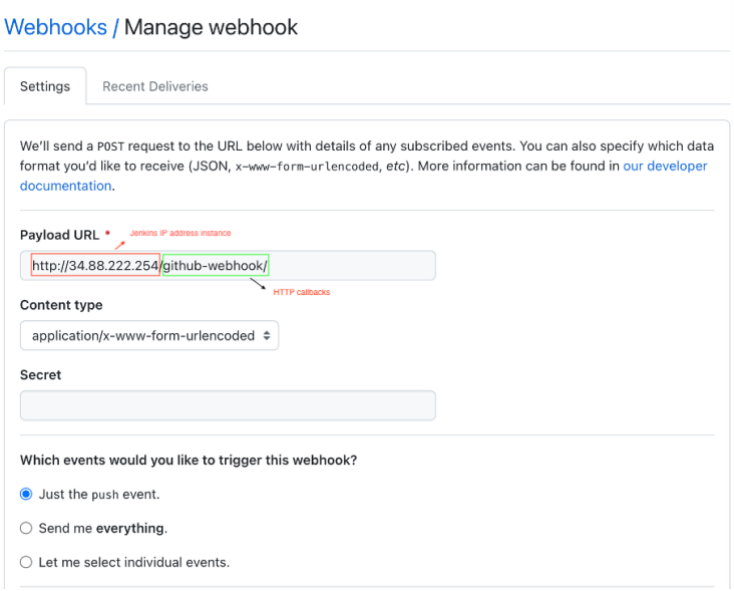

Figure 19. Webhook configuration.

The image provided illustrates the configuration of a webhook within the context of this project. By specifying the IP address of Jenkins' instance, GitHub will send a POST request whenever a developer pushes a request, thereby triggering Jenkins.

4.3 Docker and Dockerfile

The goal of the project was to implement a CI/CD pipeline, SonarQube server, and deploy a web application using containerized technology with Docker. In order to achieve this, Docker had to be installed on all instances.

To simplify the installation process, a bash script has been created. This script automates the installation of the latest versions of Docker and Docker Compose from the official website.

```
$ startup-docker.sh
#!/bin/bash
sudo apt-get update
sudo apt-get install -y \
    ca-certificates \
    curl \
    gnupg \
    lsb-release

sudo mkdir -p /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg

echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

sudo apt-get update

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

sudo systemctl enable docker.service

sudo systemctl enable containerd.service

sudo groupadd docker

sudo usermod -aG docker ${USER}

newgrp docker

sudo sysctl -w vm.max_map_count=262144
```

Figure 20. Docker auto-installation scripts.

To use the script, it is important follow these steps:

1. Pull prepared scripts.
2. Open the terminal on the Ubuntu server.

3. Execute the script using the following command: "*sudo ./startup-docker.sh*". The script will handle all the necessary steps, including updating the package lists, configuring the Docker repository, and installing Docker CLI, Docker-Compose. It also adds the current user to the docker group, allowing developers to run Docker without root privileges.

```
htrung_jobs@dev-jenkins-vm:~$ systemctl status docker
● docker.service - Docker Application Container Engine
     Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset: enabled)
     Active: active (running) since Sun 2023-07-23 15:40:52 UTC; 1min 35s ago
TriggeredBy: ● docker.socket
       Docs: https://docs.docker.com
   Main PID: 3076 (dockerd)
      Tasks: 9
     Memory: 28.5M
        CPU: 391ms
     CGroup: /system.slice/docker.service
             └─3076 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock

Jul 23 15:40:52 dev-jenkins-vm systemd[1]: Starting Docker Application Container Engine...
Jul 23 15:40:52 dev-jenkins-vm dockerd[3076]: time="2023-07-23T15:40:52.425450164Z" level=info msg="Starting u>
Jul 23 15:40:52 dev-jenkins-vm dockerd[3076]: time="2023-07-23T15:40:52.426806084Z" level=info msg="detected 1>
Jul 23 15:40:52 dev-jenkins-vm dockerd[3076]: time="2023-07-23T15:40:52.548595083Z" level=info msg="Loading co>
Jul 23 15:40:52 dev-jenkins-vm dockerd[3076]: time="2023-07-23T15:40:52.877417861Z" level=info msg="Loading co>
Jul 23 15:40:52 dev-jenkins-vm dockerd[3076]: time="2023-07-23T15:40:52.906822107Z" level=info msg="Docker dae>
Jul 23 15:40:52 dev-jenkins-vm dockerd[3076]: time="2023-07-23T15:40:52.907250333Z" level=info msg="Daemon has>
Jul 23 15:40:52 dev-jenkins-vm dockerd[3076]: time="2023-07-23T15:40:52.954304271Z" level=info msg="API listen>
Jul 23 15:40:52 dev-jenkins-vm systemd[1]: Started Docker Application Container Engine.
lines 1-21/21 (END)
^C
```

Figure 21. Verify Docker status.

The MooMoo web application is a Node.js application. The first step to containerize is building Docker images. The Dockerfile provided below shows the process.

The Dockerfile starts with "*FROM alpine*", which sets the base image for the container's runtime environment for the Node.js application. Docker offers various base images depending on user requirements. Alpine is chosen as a lightweight and efficient Linux distribution, ensuring the smallest possible image size. The next command sets the working directory path inside the container as *"/usr/src/app"*. Following that, the COPY command is used to copy the *"package.json*" file from the host to the container's working directory, and the Node.js and npm package manager are installed using the RUN command.

With the completion of image preparation, the next steps involve containerizing MooMoo. The initial step involves transferring all the application files and directories to the container's working directory. The Dockerfile exposes port 3000

to enable potential external access and sets the default command to execute upon container startup, typically "*npm start*" to initiate the Node.js application. This setup guarantees a lightweight and efficient container image that is well-suited for executing Node.js applications.

```
FROM alpine
WORKDIR /usr/src/app
COPY package.json ./
RUN apk add --update nodejs npm && npm install .
COPY . .
EXPOSE 3000
CMD ["npm", "start"]
```

Figure 22. Dockerfile for MooMoo Web Application.

4.4 SonarQube

To install SonarQube on its server (devops-sonarqube-vm), we need a Docker compose file for SonarQube is needed.

```
version: "3"

services:
  sonarqube:
    image: sonarqube:community
    depends_on:
      - db
    environment:
      SONAR_JDBC_URL: jdbc:postgresql://db:5432/sonar
      SONAR_JDBC_USERNAME: sonar
      SONAR_JDBC_PASSWORD: sonar
    volumes:
      - sonarqube_data:/opt/sonarqube/data
      - sonarqube_extensions:/opt/sonarqube/extensions
      - sonarqube_logs:/opt/sonarqube/logs
    ports:
      - "9000:9000"
  db:
    image: postgres:12
    environment:
      POSTGRES_USER: sonar
      POSTGRES_PASSWORD: sonar
    volumes:
      - postgresql:/var/lib/postgresql
      - postgresql_data:/var/lib/postgresql/data

volumes:
  sonarqube_data:
  sonarqube_extensions:
  sonarqube_logs:
  postgresql:
  postgresql_data:
```

Figure 23. Docker Compose file for SonarQube installation.

The Docker Compose file defines two containers. The SonarQube container is set up with a PostgreSQL database as its backend. SonarQube is used to store code analysis data and logs.

Once the Docker Compose file is prepared, deploying SonarQube is as simple as running the command "*docker-compose up -d*". Access to SonarQube is available at "*http://35.228.114.222:9000"* using the login details specified in the Docker file.

Integrating SonarQube with Jenkins is necessary to ensure a smooth-running pipeline. Jenkins requires SonarQube credentials to be able to do so. To obtain the token, it is necessary to go to the SonarQube's settings and generate a new token with the user type named "*jenkins*".

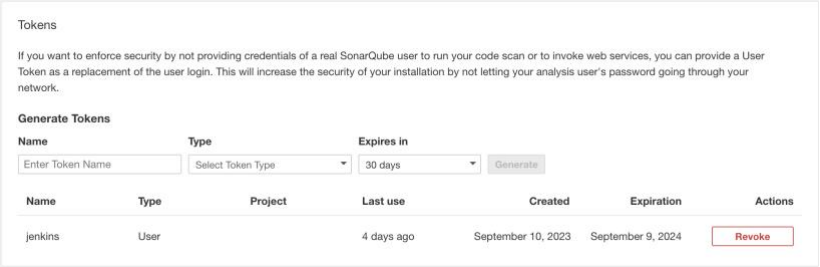

Figure 24. Generating token on SonarQube.

Additionally, SonarQube needs to have a webhook set up for every time a Jenkins job is triggered. To set up the webhook in Sonarqube, it is important to go to the project's administration settings and select "*Webhooks*". The next step is adding a new webhook with the Jenkins URL named "*jenkins-webhook*".

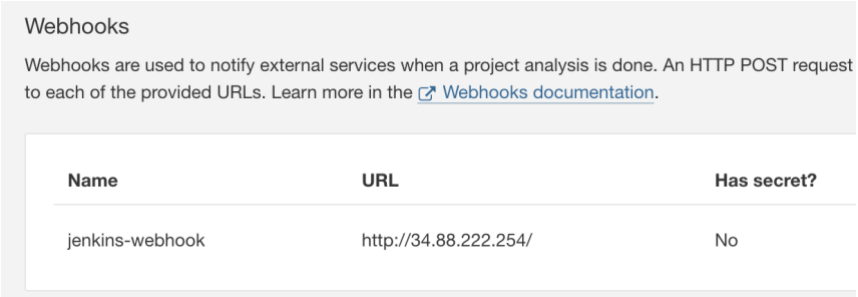

Figure 25. Setting up SonarQube webhook.

With this integration, SonarQube will provide detailed code analysis reports and quality metrics for every build in the Jenkins pipeline.

4.5 Jenkins

Docker is installed on all instances as it is easy to install Jenkins via a container on the Jenkins server (devops-jenkins-vm). By default, Jenkins is configured to run on port 8080. To further enhance security and flexibility in accessing the server, Jenkins will be installed with Nginx as a reverse proxy by defining it in the Docker-compose configuration file.

```
jenkins > docker-compose.yaml
# docker-compose.yaml
version: '3.8'
services:
  nginx:
    container_name: nginx
    restart: always
    tty: true
    image: nginx
    user: root
    volumes:
      - ./nginx:/etc/nginx/conf.d
    ports:
      - 80:80
  jenkins:
    container_name: jenkins
    restart: always
    tty: true
    image: jenkins/jenkins
    privileged: true
    user: root
    ports:
    #  - 8080:8080
      - 50000:50000
    volumes:
      - jenkins:/var/jenkins_home
      - /var/run/docker.sock:/var/run/docker.sock
volumes:
  jenkins:
```

Figure 26. Docker compose file to install Jenkins and Nginx.

The Docker-compose file contains configurations for Jenkins and Nginx services. Nginx is set up to utilize the official Nginx Docker image and expose port 80 for HTTP traffic. The Jenkins service runs with root privileges and is accessible within the container on port 50000. It also includes volume mounts to ensure the

persistence of the Jenkins data and to enable communication with the host's Docker daemon. Both services are configured to automatically restart when Docker is online. Additionally, Nginx configuration is required to redirect user requests to the Jenkins server inside the Docker container.

Figure 27. How Nginx forward user request to Jenkins.

After successfully installing Jenkins, the next step is to configure it. Developers can directly access Jenkins by using the external IP address of the instance (devops-jenkins-vm - 34.88.222.254). It is important to note that Jenkins requires an initial password, which is located within the Jenkins container.

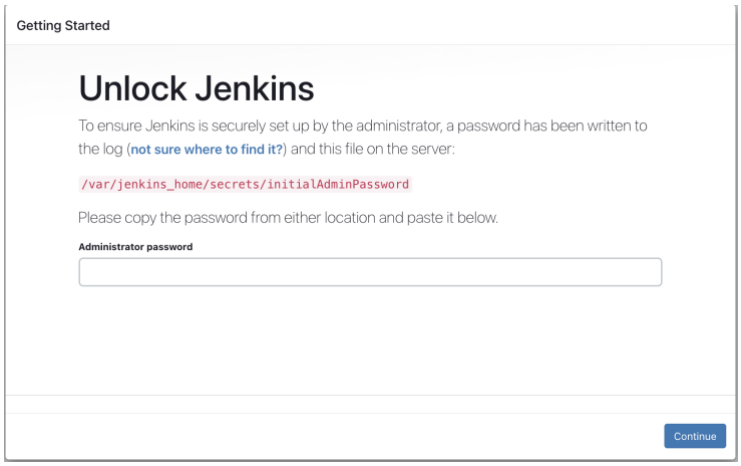

Figure 28. Setting up Jenkins.

Jenkins needs to install the required plugins and sets up an administrator password. After completing the setup stage, Jenkins is ready to use.

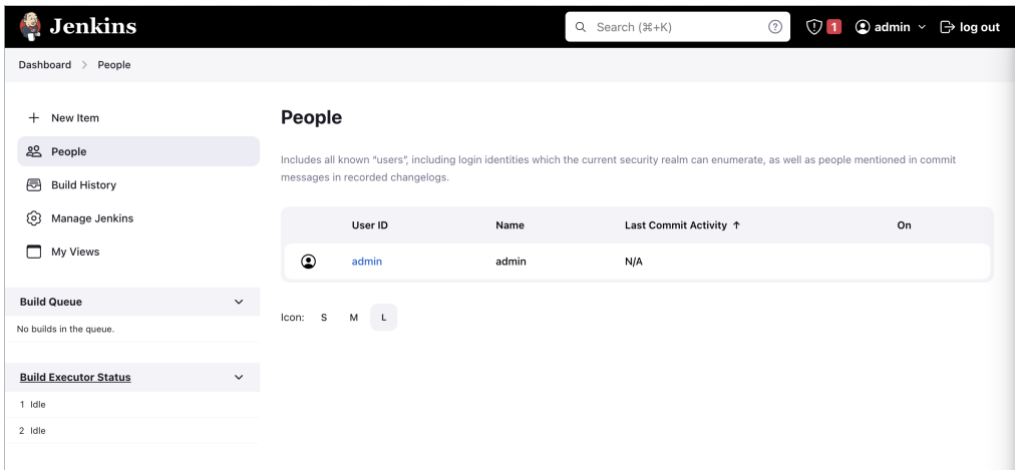

Figure 29. Jenkins dashboard.

Next, we need to specify credentials for tools that will be used in this project. The types of credentials are as followed:

- SonarQube: Login Credential
- Docker Hub: Login Credential
- Telegram: chatID and Token
- Production Environment: SSH key

With these configurations, Jenkins not only integrates with tools in the CI/CD pipeline but also ensures the security and reliability of interactions with external services.

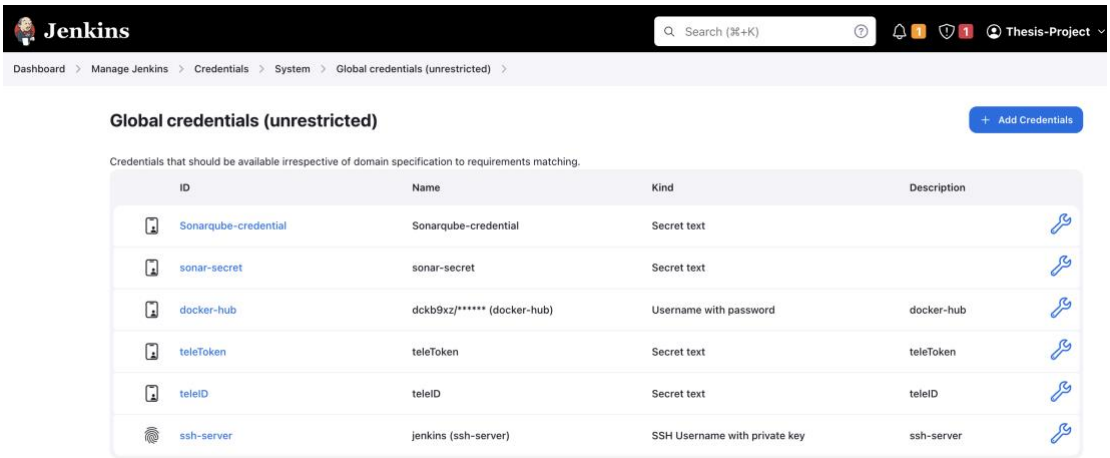

Figure 30. Setting up credentials on Jenkins.

4.6 Configuring CI/CD Pipeline

Developers can start designing their CI/CD pipeline in Jenkins by creating a new project from the dashboard. Among the various project types available in Jenkins, the ‘Pipeline’ project type is the most suitable and advantageous option considering the project's scope. The new project is created with named “movie-pipeline”.

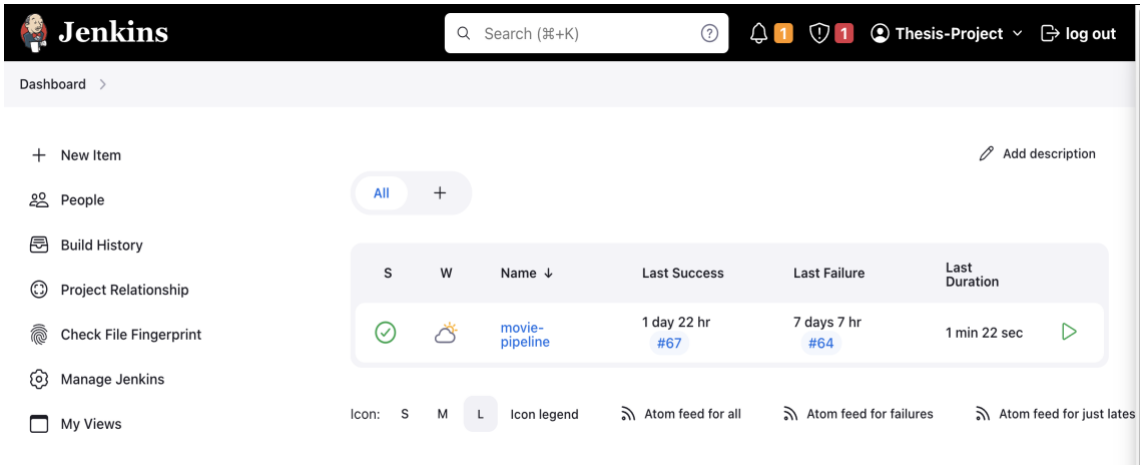

Figure 31. Jenkins pipeline overview.

Next, configure Jenkins to receive requests from GitHub webhook. This will ensure that any code changes pushed to GitHub will trigger the “movie-pipeline” job. In the Build Trigger section, select “*GitHub hook trigger for GITScm polling*”. In the pipeline section, fill in the code repository URL and set the pipeline to build the project on the main branch. With these settings in place, Jenkins will be able to receive and respond to GitHub webhook requests, and the automated pipeline will be created.

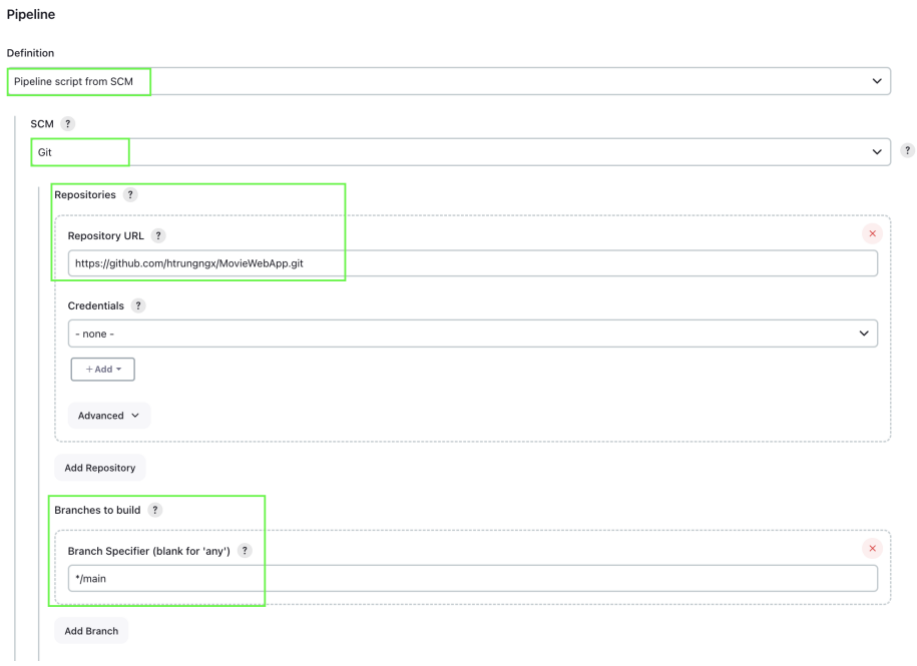

Figure 32. Jenkins pipeline configuration.

At this point, Jenkins will call GitHub to trigger the first build. The result can be seen in the picture below.

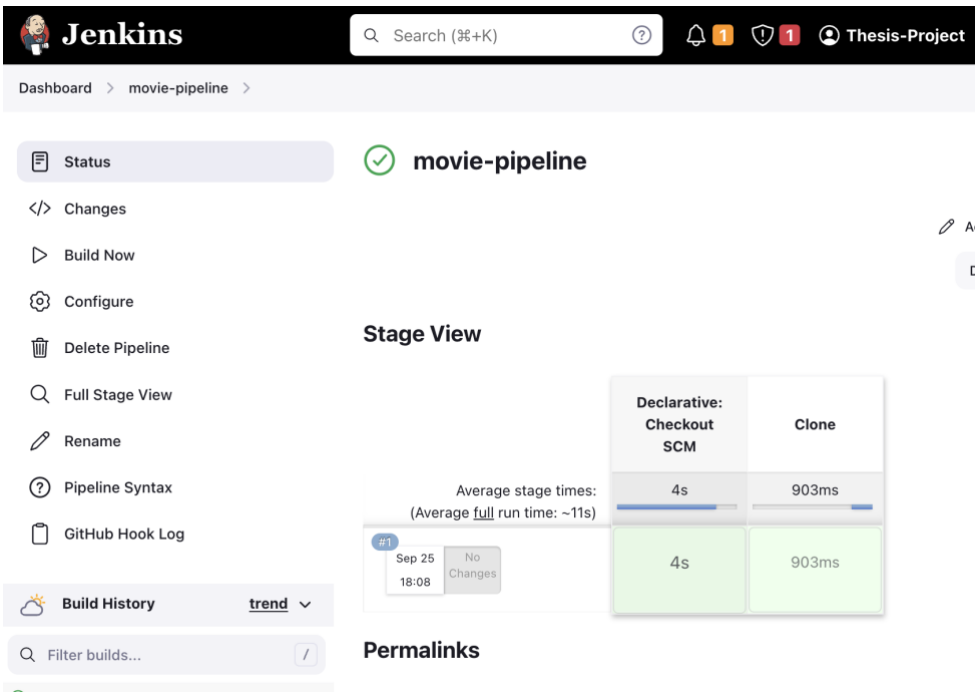

Figure 33. 1st Jenkins build

Jenkins allows developers to customize their CI/CD flow by using Jenkinsfile - which works as as a blueprint for seamless integration and deployment. Developers can define as many steps as they want from installing dependencies with "*npm install*" command, packaging and uploading images with Docker to delivering the application to any servers. All modifications to the pipeline can be viewed through the Jenkins dashboard.

```
Jenkinsfile M ×
Jenkinsfile
pipeline {
    agent any
    environment {
        //Tele-credential
        TOKEN = credentials('teleToken')
        CHAT_ID = credentials('teleID')

        //Git info
        GIT_INFO = "Branch: ${GIT_BRANCH}\nLast Message: ${GIT_MESSAGE}\nAuthor: ${GIT_AUTHOR}\nCommit: ${GIT_COMMIT_SHORT}\nBuild Number: #${CURRENT_BUILD_NUMBER}"
        CURRENT_BUILD_NUMBER = "${currentBuild.number}"
        GIT_MESSAGE = sh(returnStdout: true, script: "git log -n 1 --format=%s ${GIT_COMMIT}").trim()
        GIT_AUTHOR = sh(returnStdout: true, script: "git log -n 1 --format=%ae ${GIT_COMMIT}").trim()
        GIT_COMMIT_SHORT = sh(returnStdout: true, script: "git rev-parse --short ${GIT_COMMIT}").trim()

        PRE_BUILD = "Jenkins is starting 🚀\n ${LINE_SKIP}\n${GIT_INFO}\n ${LINE_SKIP}\n [${JOB_NAME}] is starting to build"

        LINE_SKIP = "-----------------------------------------------"

        TEXT_SUCCESS_BUILD = " ✅ [${JOB_NAME}] is successfully built"
        TEXT_FAILURE_BUILD = " ❌ [${JOB_NAME}] built FAILED"
    }
    tools {
        jdk 'jdk20'
        nodejs 'nodejs20'
    }
```

Figure 34. Extract of Jenkinsfile.

The Jenkinsfile begins by configuring environment variables for Telegram integration. These variables, such as "*CHAT_ID*" and "*TOKEN*", are essential for sending notifications. The picture shows that there are predefined messaging templates that will inform developers about the state of the pipeline job. For example, the "*PRE_BUILD*" variable sends a notification at the start of the job, including information about the Git branch, recent commit message, build number, and author. Additionally, developers need to specify the required package manager, such as Nodejs, which is necessary for the MooMoo app or the Java Development Kit (JDK) for compiling, debugging, and running Java applications, specifically Jenkins, which is built on Java.

```
stages {
        stage('Pre-Build') {
            steps {
                sh "curl --location --request POST
'https://api.telegram.org/bot${TOKEN}/sendMessage' --form text='${PRE_BUILD}' --form
chat_id='${CHAT_ID}'"
            }
        }
        //Clone code from repogistory
        stage('Clone') {
            steps {
                git branch: 'main',
                    url: 'https://github.com/htrungngx/MovieWebApp.git'
            }
        }
        //Send code to Analysis Server
        stage('Code Analysis') {
            environment {
                scannerHome = tool 'Sonar-scanner'
            }
            steps {
                withSonarQubeEnv('Sonarqube') {
                    sh """
                        ${scannerHome}/bin/sonar-scanner \
                        -Dsonar.projectKey=Jenkins-Testing \
                        -Dsonar.sources=.
                    """
                }
            }
        }
        //Build code with Docker and Push to Registry
        stage('Build Image') {
            steps {
                withDockerRegistry(
                    credentialsId: 'docker-hub',
                    url: 'https://index.docker.io/v1/'
                ) {
                    sh 'docker build -t dckb9xz/app .'
                    sh 'docker push dckb9xz/app'
                }
            }
        }
        //Deploy to Dev Environment
        stage('Deploy to DevEnv') {
            steps {
                echo 'Deploying and Cleaning'
                sh 'docker pull dckb9xz/app:latest'
                sh 'docker ps -a'
                sh "docker rm -f movieapp || echo 'The container does not exist'" //Remove
exist container
                sh 'docker run --name movieapp -p 3000:3000 -d dckb9xz/app'
            }
        }
        //Deploy to production server
        stage('SSH to Production') {
            steps {
                sshagent(['ssh-server']) {
                    sh """
                    ssh -o StrictHostKeyChecking=no htrung_jobs@34.88.164.172 < ssh-scripts.sh
                    """
                }
            }
        }
    }
```

Figure 35. Extract of stage in Jenkinsfile

As can be seen in the picture above, there are five stages in the main of Jenkinsfile. Each stage plays a vital role in ensuring code quality, containerization, and deployment.

- **Pre-Build** – Notifications are sent to developers via Telegram. The message includes contextual information such as branch name, build number, and commit message.
- **Clone** – The source code from the GitHub repository will be plugged in and cloned to ensure that the most up-to-date version of the web application is used for subsequent stages.
- **Code Analysis** – The environment is configured to send code to SonarQube to scan for any quality and security issues.

- **Build Image** – This step focuses on containerization. The codebase is transformed into a Docker image through a series of shell commands. The Docker image is then pushed to a Docker Registry.
- **Deploy to Dev Environment** – The latest Docker image from the registry will be retrieved. A container named "*movieapp*" is started with port mapping, allowing the web to be accessed on port 3000.
- **Deploy to Production Server** – In this final stage, Jenkins uses SSH to secure access to the server (devops-app-vm) to pull and run a web container image. The user can access to MooMoo at the instance IP's address (34.88.164.172)

Lastly, the status of the Jenkins pipeline will be sent to Telegram. This final step ensures that developers are informed about the outcome and overall status of the pipeline.

```
post {
            success {
                script {
                    sh "curl --location --request POST
'https://api.telegram.org/bot${TOKEN}/sendMessage' --form text='${TEXT_SUCCESS_BUILD}' --form
chat_id='${CHAT_ID}'"
                }
            }
            failure {
                script {
                    sh "curl --location --request POST
'https://api.telegram.org/bot${TOKEN}/sendMessage' --form text='${TEXT_FAILURE_BUILD}' --form
chat_id='${CHAT_ID}'"
                }
            }
        }
}
```

Figure 36. Extract of notification part in Jenkinsfile.

4.7 Monitoring Solution

Monitoring is an important part of a CI/CD pipeline. The next step is monitoring Jenkins servers to make sure it runs smoothly without any possible downtime.

Similarly, as installing SonarQube, Prometheus and Grafana will be installed by using the docker-compose file.

In the Docker Compose file, we configure Prometheus to be accessible externally on port 9090. We also explicitly define the location of Prometheus's configuration file as "*/etc/prometheus/prometheus.yml*" On the other hand, Grafana is made accessible via port 3001 and is defined to create an administrator account. To simplify the management of Grafana-related data, such as dashboard templates, user data, and configurations, we create a volume named "*./grafana*". Both services are configured with a restart policy to ensure that they automatically start when Docker is started.

```
services:
  prometheus:
    image: prom/prometheus
    container_name: prometheus
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'
    ports:
      - 9090:9090
    restart: unless-stopped
    volumes:
      - ./prometheus:/etc/prometheus
      - prom_data:/prometheus
  grafana:
    image: grafana/grafana
    container_name: grafana
    ports:
      - 3001:3000
    restart: unless-stopped
    environment:
      - GF_SECURITY_ADMIN_USER=admin
      - GF_SECURITY_ADMIN_PASSWORD=grafana
    volumes:
      - ./grafana:/etc/grafana/provisioning/datasources
volumes:
  prom_data:
```

Figure 37. Docker compose for Monitoring Stack installation.

In order to start these services, execute the command *"docker compose up -d"*. After a successful installation, developers can connect to Prometheus via "*http://34.88.222.254:9090*"

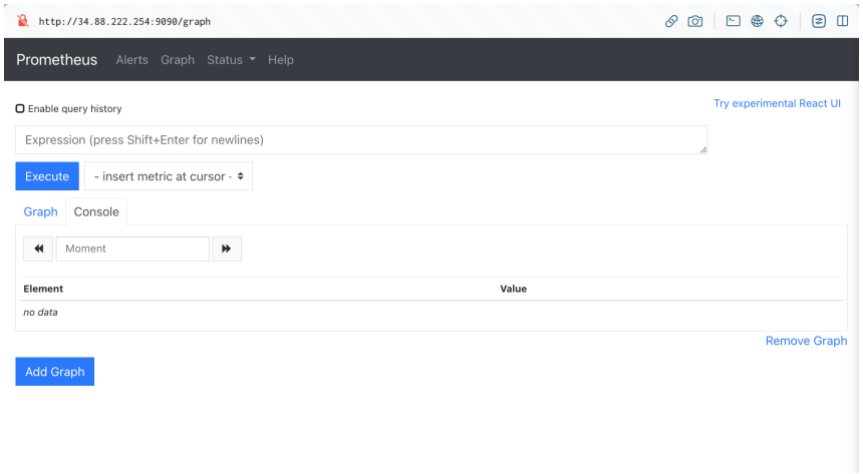

Figure 38. Prometheus dashboard.

To collect Jenkins metrics, integration of Prometheus with Jenkins is required by installing its plugins and configuring it as follows:

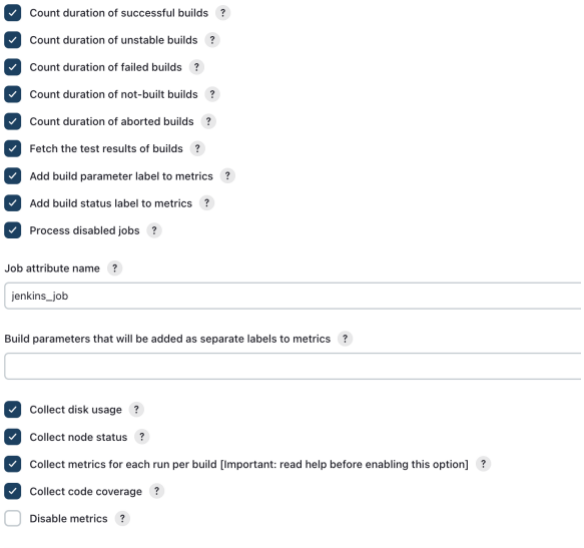

Figure 39. Jenkins configuration to collect metrics.

Prometheus gathers crucial metrics from Jenkins, such as build metrics, and pipeline status. To send this data to Prometheus, Jenkins's URL needs to be specified in the Target section of Prometheus's settings. When the status turns green, it indicates that the connection has been established.

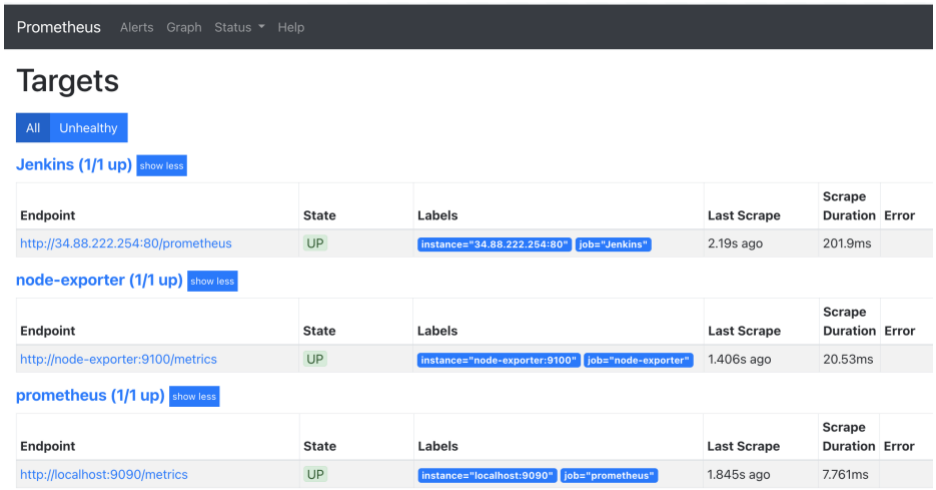

Figure 40. Prometheus's target.

Prometheus is now gathering unprocessed information from Jenkins. To visualize the metrics, access Grafana at "*http://34.88.222.254:3001*" using the login details specified in the Docker compose file. In the Grafana settings, add Prometheus as a data source and configure its server URL (in this case: http://34.88.222.254:9090). Once this is successfully completed, Grafana will be able to utilize the data collected by Prometheus.

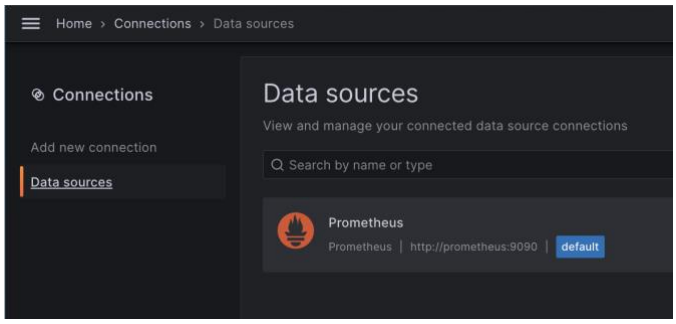

Figure 41. Adding data source to Grafana.

After completing the setup, click on the 'Add panel' button to begin creating the dashboard. For each panel, we will need to configure queries from the data source to collect the desired metrics. For instance, to obtain the number of active plugins, use the query "*jenkins_plugins_active{}*", and then select the appropriate chart type for visualization. Panels can be grouped together to observe them

better. The following figures (Figure 42 and Figure 43) illustrate a sample Jenkins and the system dashboard utilized in this project.

Figure 42. Jenkins monitoring dashboard.

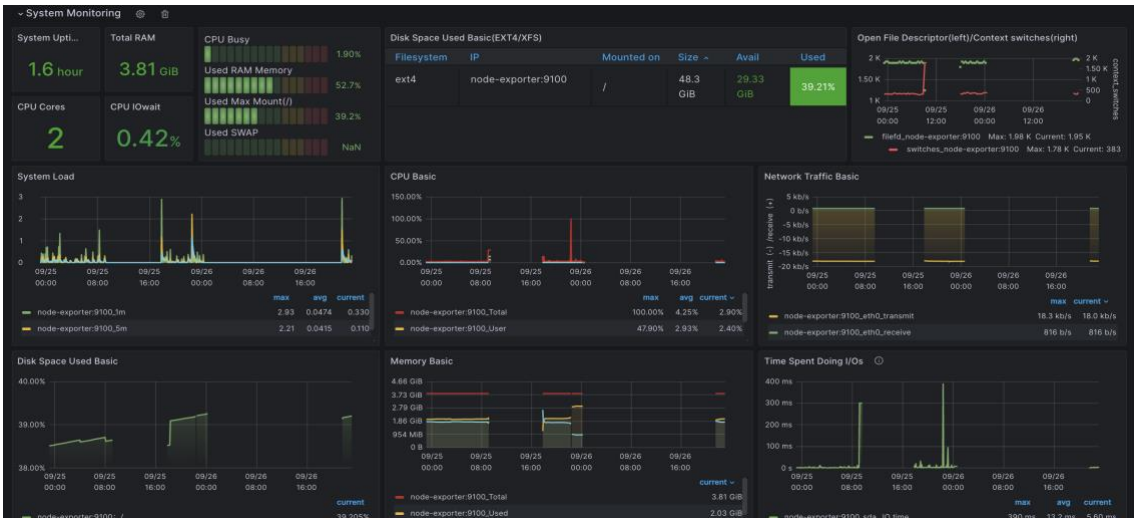

Figure 43. Jenkins's instance monitoring dashboard.

5 Results and Challenges

5.1 Project Results

In this section, we will thoroughly evaluate the performance of the CI/CD pipeline implemented for the MooMoo web application and discuss challenges and potential updates for further development and optimization.

First of all, as mentioned in the thesis title, the project showcases a comprehensive web application life cycle. The MooMoo application is containerized, analyzed, and deployed on both testing and production environments. As demonstrated in the picture below, the entire project goes through 7 stages and finishes within 3 minutes. This achievement allows for the rapid delivery of new features and security patches to users, ensuring that they can benefit from the latest improvements without any delay.

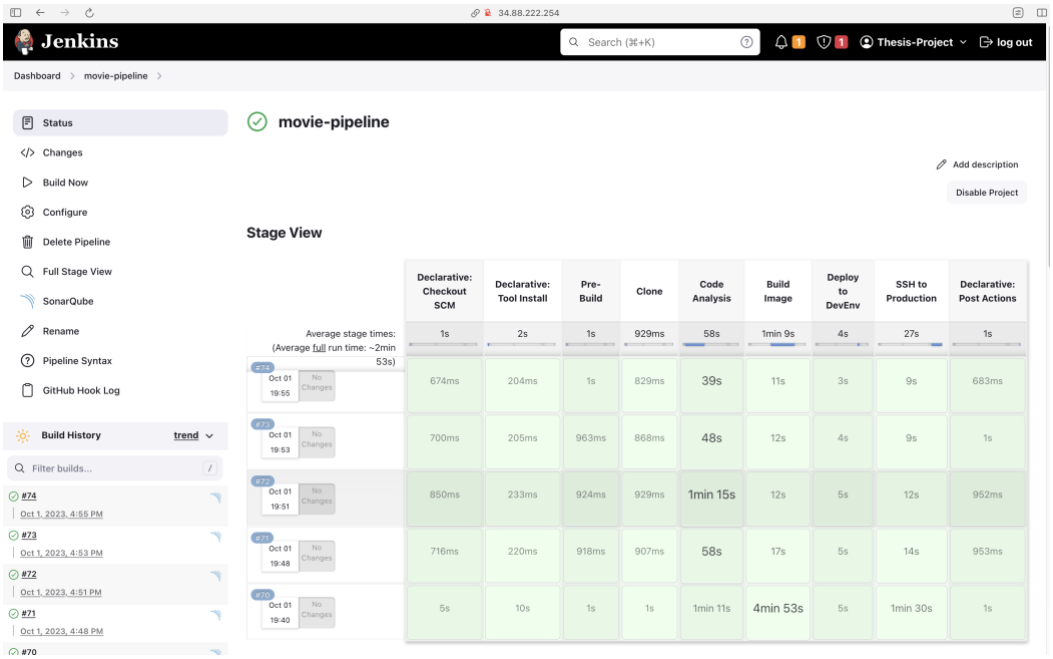

Figure 44. Jenkins dashboard.

After running the pipeline, Jenkins will inform the developer about the pipeline's status. The figure below shows an extract of the notification in both the success/fail state sent by Jenkins pipeline for each build. Jenkins sends to the

developer important messages such as branch, message commit, build number and the status of pipeline after running the job.

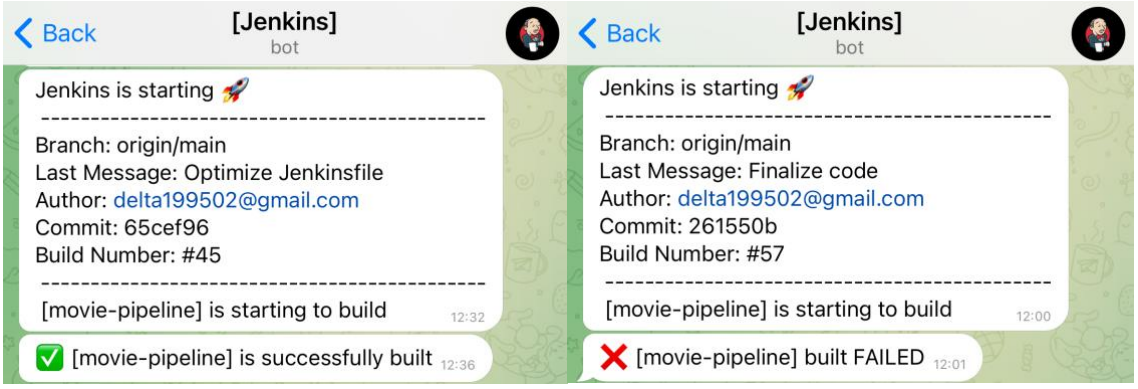

Figure 45. Telegram notification about Jenkins pipeline.

Secondly, by containerizing projects, the developer team can work in parallel without concerns about resolving dependencies or working under different environments. This enhances collaboration, reduces debugging time, and allows developers to focus more on writing code.

Thirdly, the entire project was hosted on Google Cloud and powered by three virtual machines. By applying infrastructure as code, developers can manage resources in a structured manner. This ensures that the configuration remains consistent across deployments and can be started in just a few minutes. Additionally, all tools have their own configuration files. This modularity not only allows developers to reuse configurations for similar projects but also reduces troubleshooting and setup time.

5.2 Project Challenges

Throughout the planning and implementation of this project, the author faced several challenges. One of the most difficult challenges was the complexity of the project. The pipeline involved integrating multiple tools, ranging from CI/CD automation to cloud deployment and monitoring. Each of these tools has its own documentation that requires careful research before planning and implementation into this project. Despite having an IT background and being

familiar with learning new technologies, it was still challenging to integrate and apply all the acquired knowledge in the project.

Combining multiple tools often leads to frequent errors, which is the second issue that needs to be addressed. The CI/CD pipeline integrates multiple tools into one and working across multiple environments simultaneously. Hence, errors become more complex and come from various sources. The more tools involved, the more complex the problems become. Tracing and troubleshooting these errors require time and a solid understanding of how to use them. Additionally, the DevOps mindset is still relatively new and not widely applied in practical projects. The shortage of developers working on similar projects and utilizing the same tools results in a lack of support and instructions.

5.3 Further Improvements

While the project is currently small in scale and there are many additional features that can be implemented in the pipeline, further development is grouped into the table below.

Features	Description
Logging solution for whole pipeline	Better observation, easy to maintain
Alert managing	Configured alerts can be notified when specific events happen
Web features	Improved UI Search bar AI integrated
Integrate testing. process into pipeline	Unit testing

Table 3: Additional features.

6 Conclusion

In conclusion, this main goal of this study was to depict a big picture of DevOps culture in software development and to implement a CI/CD pipeline for a web application. Throughout the thesis, various research and concepts have been studied to fully understand the central concept of this field.

The Overview has explained the term DevOps and related practices. It can be concluded that DevOps empowers software development by using CI/CD and containerization technology. It also enables developers to automate their work, improve collaboration, and deliver applications to users faster.

In addition, a summary of all the tools and methods used was given in Section 3 (Methodology). With a comprehensive explanation, for example of Jenkins, SonarQube, Terraform and CI/CD pipeline, the framework for configuration steps was set. Moreover, this section went through the structure of the MooMoo web application.

Next, the implementation section focused on tool configuration for MooMoo's development pipeline. This phase included setting up the CI/CD tool (Jenkins), code analysis process (SonarQube), cloud infrastructure (Google Cloud, Terraform), containerization (Docker) and monitoring stack (Grafana, Prometheus and Node exporter). Despite numerous challenges when integrating multiple tools seamlessly, the challenges were successfully overcome and valuable experiences was gained.

Regarding the project outcomes, there are several achievements to be highlighted. Firstly, the pipeline was executed successfully, taking less than three minutes to deliver software to both testing and production environments. The pipeline started by pulling code from the code repository, which then pad the analysis process before being containerized with the latest tag. The code is deployed to both environments and monitored every minute. Moreover, Google Cloud and Terraform played a crucial role in this project, providing the benefits of

cloud computing in software development. The entire project is hosted on three cloud instances and Terraform was successfully integrated to oversee the cloud infrastructure.

Additionally, the project accomplished the goal of reusability, as some parts of the configuration files are reusable. The modular design of the project allows for the possibility of reconfiguration and reuse in other projects as well. If the project scales up in the future, the CI/CD flow can be reused with small modifications. However, there are still plenty of improvements that need to be addressed in the future, such as incorporating the major features of the MooMoo web application or integrating logging for each instance.

Finally, the purpose of the project was to help the author gain a wide range of knowledge and experience of DevOps and Cloud. The author also gained insights into the importance of the application lifecycle and the DevOps approach in software development. In sum, this project serves as a foundation for future endeavors in applying CI/CD in software development and opens the potential opportunities to dive deeper into DevOps and Cloud Computing.

Bibliography

1.Atlassian. What is DevOps? | Atlassian [Internet]. Atlassian. 2016 [cited 2023 Oct 9]. Available from: https://www.atlassian.com/devops

2.Freeman E. DevOps. Hoboken, Nj: John Wiley & Sons, Inc; 2019.

3.Atlassian. Benefits of DevOps [Internet]. Atlassian. 2023. Available from: https://www.atlassian.com/devops/what-is-devops/benefits-of-devops

4.Atlassian. DevOps Culture [Internet]. Atlassian. Available from: https://www.atlassian.com/devops/what-is-devops/devops-culture

5.Beattie T, Hepburn M, O'Connor N, Spring D. DevOps Culture and Practice with OpenShift. Packt Publishing Ltd; 2021.

6.How CI/CD Can Save App Development Time and Create Robust Apps [Internet]. TechAhead. 2023. Available from: https://www.techaheadcorp.com/blog/how-ci-cd-save-app-development-time/

7.Redhat. What is CI/CD? [Internet]. Redhat.com. 2019. Available from: https://www.redhat.com/en/topics/devops/what-is-ci-cd

8.Gitlab. What is a CI/CD pipeline? [Internet]. about.gitlab.com. Available from: https://about.gitlab.com/topics/ci-cd/cicd-pipeline/

9.Belmont JM. Hands-On Continuous Integration and Delivery. Packt Publishing Ltd; 2018.

10.Amazon. The CI/CD litmus test: Is your pipeline fully CI/CD? [Internet]. docs.aws.amazon.com. Available from: https://docs.aws.amazon.com/prescriptive-guidance/latest/strategy-cicd-litmus/introduction.html

11.Homepage | Travis CI – Start building today! [Internet]. Travis CI. Available from: https://www.travis-ci.com/

12.GitHub. Features • GitHub Actions [Internet]. GitHub. Available from: https://github.com/features/actions

13.Jenkins. Jenkins [Internet]. Jenkins. 2023. Available from: https://www.jenkins.io/

14.Google. What are containers? [Internet]. Google Cloud. Available from: https://cloud.google.com/learn/what-are-containers

15.Docker. Docker overview [Internet]. Docker Documentation. 2020. Available from: https://docs.docker.com/get-started/overview/

16.Sonar. SonarQube 10.1 [Internet]. docs.sonarsource.com. Available from: https://docs.sonarsource.com/sonarqube/latest/

17.Prometheus. Overview | Prometheus [Internet]. Prometheus.io. 2012. Available from: https://prometheus.io/docs/introduction/overview/

18.Grafana. Grafana® Features [Internet]. Grafana Labs. Available from: https://grafana.com/grafana/

19.Prometheus. Monitoring Linux host metrics with the Node Exporter | Prometheus [Internet]. prometheus.io. Available from: https://prometheus.io/docs/guides/node-exporter/

20.HashiCorp. Terraform by HashiCorp [Internet]. Terraform by HashiCorp. 2023. Available from: https://www.terraform.io/

21.Hashicorp. What is Terraform | Terraform | HashiCorp Developer [Internet]. What is Terraform | Terraform | HashiCorp Developer. Available from: https://developer.hashicorp.com/terraform/intro

22.IBM. What is cloud computing | IBM [Internet]. www.ibm.com. 2023. Available from: https://www.ibm.com/topics/cloud-computing

23.Google. Products & Services [Internet]. Google Cloud. Available from: https://cloud.google.com/products

24.Canalys. Canalys Newsroom - Global cloud services market growth slows to 16% in Q2 2023 [Internet]. www.canalys.com. [cited 2023 Oct 9]. Available from: https://www.canalys.com/newsroom/global-cloud-services-q2-2023

25.Statista. Google Cloud revenue 2020 [Internet]. Statista. Available from: https://www.statista.com/statistics/478176/google-public-cloud-revenue/

26.JavaPoint. ReactJS Introduction [Internet]. www.javatpoint.com. Available from: https://www.javatpoint.com/react-introduction

27.Singh S. Virtual DOM and Real DOM: Understanding the Differences [Internet]. Medium. 2023. Available from: https://medium.com/@surksha8/virtual-dom-and-real-dom-understanding-the-differences-da8f3fab4261

28.Google. Machine families resource and comparison guide | Compute Engine Documentation [Internet]. Google Cloud. Available from: https://cloud.google.com/compute/docs/machine-resource

29.GitHub. Webhooks documentation [Internet]. GitHub Docs. Available from: https://docs.github.com/en/webhooks