# Project Overview

* 1. Project Title: End To End CICD Pipeline Project with Jenkins
  2. Summary:

A screenshot of a computer

Description automatically generated  
This project implements a robust CI/CD pipeline powered by Jenkins to automate the build, test, and deployment processes. The pipeline begins with code development and version control using a centralized repository like GitHub.

Jenkins orchestrates the entire pipeline, initiating builds triggered by code commits. Maven is employed to compile the project, while SonarQube analyzes the code for quality and security issues. Upon successful code quality checks, Docker images are constructed using Dockerfiles and pushed to AWS ECR.

The deployment process is automated, with the Docker images being deployed to staging and production environments. This ensures a seamless and efficient delivery of the application, minimizing manual intervention and reducing the risk of human error. By leveraging Jenkins, this pipeline optimizes the development and deployment workflow, enabling faster and more reliable software delivery.

* 1. Purpose and Goals:

The primary purpose of this project is to establish a highly efficient and automated CI/CD pipeline using Jenkins. This pipeline aims to streamline the development, testing, and deployment processes, ensuring rapid and reliable delivery of high-quality software.

The specific goals of the project are:

* Reduce manual intervention and minimize human error by automating the build, test, and deployment steps.
* Integrate SonarQube to analyze code for potential issues, ensuring adherence to coding standards and best practices.
  1. Prerequisites:
* AWS Services:
  + AWS EC2
  + AWS ECR
  + Security Groups
  + IAM Role
* CI/build Server:
  + Docker
  + Java 17
  + Apache Maven
  + Git
  + Sonar Scanner plugin
  + Jenkins
* Sonarqube Server:
  + Docker
  + SonarQube
  + Webhook
* CD Server:
  + Docker
  + AWS CLI
  + Java 17

# Implementation

* 1. To initiate the build process, an Amazon Linux 2 instance of type t2.medium is provisioned. This instance will serve as the build server. The server is configured with either a newly generated key pair or an existing one, and a security group is established to allow SSH connections. Once the instance is successfully launched, a connection is established to the server's terminal using MobaXterm. The connection is made by specifying the public IP address of the instance, the default username (ec2-user), and the corresponding private key file.

A screenshot of a computer

Description automatically generated  
A screenshot of a software

Description automatically generated  
A screenshot of a computer

Description automatically generated  
A screenshot of a computer

Description automatically generated  
A screenshot of a computer screen

Description automatically generated  
A screenshot of a computer

Description automatically generated  
A computer screen with text and numbers

Description automatically generated

* 1. To prepare the build server for the build process, it's necessary to elevate privileges to root using **sudo su -**. This allows for the installation of additional packages. The **epel** repository, which provides a vast array of software packages, is installed to expand the available options. Subsequently, the **git** package is installed, enabling the retrieval of source code from Git repositories. Finally, **Java 17** is downloaded and installed from the official documentation, ensuring that the necessary runtime environment is available for building Java-based applications.

A screenshot of a box

Description automatically generated

* 1. To equip the build server with the necessary tools for building Java projects, **Apache Maven** is installed. The latest version of Maven, currently 3.9.9, is downloaded from the official Maven repository. The downloaded tar file is extracted to the **/opt** directory, a common location for installing software on Linux systems. We won’t be specifying the path here, as the path will be specified in Jenkins. Following the installation of Maven, **Docker** is installed on the server. Docker, a containerization platform, is essential for packaging and deploying applications.

A screenshot of a computer

Description automatically generated

* 1. After ensuring Maven and Docker are set up, we'll proceed to install Jenkins. By searching for "**Jenkins linux rpm**" on a search engine and following the initial four commands from the top result (excluding the Java 17 installation, as we've already installed it), we can successfully establish the Jenkins environment on our build server.

A screenshot of a web page

Description automatically generated  
A screenshot of a computer program

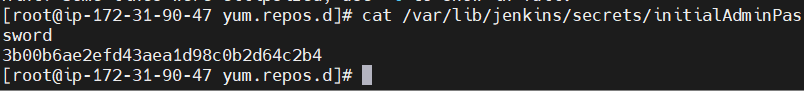
Description automatically generated  
A computer screen shot of a program

Description automatically generated  


* 1. To access the newly installed Jenkins instance, we'll need to configure the build server's security group to allow inbound traffic on port 8080 from any source. Once the security group is adjusted, we can retrieve the build server's public IP address and access the Jenkins interface by entering **<IP\_address>:8080** in a web browser. This will direct us to the Jenkins getting started page, where we'll find instructions to obtain the initial administrator password using the **cat <given\_path>** command. After inputting the password, we can proceed to the Jenkins customization page and select **"Install suggested plugins."** Finally, we'll create an administrative user by filling out the required information.

A screenshot of a computer

Description automatically generated  
A screenshot of a computer

Description automatically generated  
  
A screenshot of a computer

Description automatically generated  
A screenshot of a computer

Description automatically generated

* 1. To configure Maven within Jenkins, navigate to **"Manage Jenkins"** and select **"Tools."** Scroll down to the **"Maven Installations"** section and add a new Maven installation. Specify a name for the installation (e.g., the Maven version) and uncheck the "Install automatically" option. Manually input the correct path to the Maven installation directory on the build server. Finally, apply and save the configuration.

A white rectangular object with a white background

Description automatically generated  
A close-up of a sign

Description automatically generated  
A screenshot of a chat

Description automatically generated

* 1. To initiate the build process, we'll create a new pipeline item in Jenkins. After selecting the **"Pipeline"** type, we'll opt for a declarative pipeline approach, which typically begins with the keyword **"pipeline."** Within the pipeline script, we'll declare the necessary tools, including the previously configured Maven installation. The pipeline will then proceed through stages: first, checking out or cloning the project from its source repository to the build server's workspace, followed by building the project using Maven. Once the pipeline is saved and triggered, we can monitor its progress and verify the successful build. Upon completion, the built artifact (JAR file) will be located in the project's target directory within the Jenkins workspace.

A screenshot of a computer

Description automatically generated  
A screenshot of a computer

Description automatically generated  
A white rectangular object with a white stripe

Description automatically generated  
A screen shot of a computer

Description automatically generated  
A screen shot of a computer

Description automatically generated

* 1. To set up the SonarQube server, we'll follow a similar process as the build server. We'll launch a new Ubuntu instance on AWS, using the t2.medium instance type. Once the instance is up and running, we can access it via MobaXterm.

A screenshot of a computer

Description automatically generated  
A screenshot of a computer program

Description automatically generated  
A screenshot of a computer

Description automatically generated  
A screenshot of a computer

Description automatically generated  
A screenshot of a computer

Description automatically generated  
A screenshot of a computer

Description automatically generated

* 1. To initiate the SonarQube service, we'll first consult the SonarQube documentation to identify the latest version. Subsequently, we'll install Docker on the SonarQube server, as we intend to deploy SonarQube as a Docker container for streamlined setup. To launch the SonarQube container, we'll execute a Docker command that assigns a name to the container, exposes port 9000 for SonarQube access, and utilizes the latest SonarQube image to run the container.

A screenshot of a computer

Description automatically generated  
  
A black background with white text

Description automatically generated

* 1. To access the SonarQube dashboard, we'll need to configure the security group of the SonarQube server to allow inbound traffic on port 9000. Once the security group is adjusted, we can retrieve the SonarQube server's public IP address and access the SonarQube interface by entering http://<IP\_address>:9000 in a web browser. This will direct us to the SonarQube dashboard.

A screenshot of a computer

Description automatically generated  
A screenshot of a computer

Description automatically generated

* 1. To enable seamless communication between the build server and the SonarQube server, we'll need to generate a SonarQube token. This token can be obtained by accessing the user profile, navigating to the security settings, and generating a new token. Once generated, we'll save the token for later use. Returning to the Jenkins dashboard, we'll navigate to **"Manage Jenkins"** and proceed to the **"Plugins"** section. Here, we'll search for and install the **SonarQube Scanner** plugin. Subsequently, we'll create a new credential of type **"Secret text"** and paste the generated SonarQube token into the secret field. For the ID, we can assign any name, but it's recommended to use the same name as the token for better clarity.

A screenshot of a computer

Description automatically generated  
A white rectangular object with a black stripe

Description automatically generated  
A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated  
**A white background with black and white clouds

Description automatically generated  
A screenshot of a computer

Description automatically generated  
A close-up of a screen

Description automatically generated  
A screenshot of a computer

Description automatically generated**

* 1. To configure SonarQube within Jenkins, we'll navigate to the "System Configuration" section. Under "SonarQube Installations," we'll add a new SonarQube server instance. We'll assign a name to the instance, such as "sonar-<version>" to reflect the SonarQube version being used. Next, we'll input the server URL, which consists of the SonarQube server's public IP address and port number (e.g., http://<IP\_address>:9000). Finally, we'll associate the previously created SonarQube credentials with this server instance.

A close-up of a white background

Description automatically generated  
A white background with a black and white background

Description automatically generated with medium confidence  
A screenshot of a computer

Description automatically generated

* 1. To integrate SonarQube into our build pipeline, we'll incorporate a SonarQube Scanner step. This step will leverage the SonarQube server and credentials we've configured to analyze the built artifact. We'll search for the appropriate SonarQube Scanner command and modify it to reference the specific SonarQube server instance and credentials we've set up. After updating the pipeline script, we'll trigger a new build. If the build is successful, we can then access the SonarQube dashboard to review the project's quality gate status, which will indicate whether the project has passed or failed the quality checks.

A screenshot of a browser

Description automatically generated

A screenshot of a computer

Description automatically generatedA screenshot of a computer program

Description automatically generatedA screenshot of a computer

Description automatically generatedA close up of words

Description automatically generatedA screenshot of a computer

Description automatically generated

* 1. The final step in the SonarQube integration process involves configuring Jenkins to recognize the analysis results from SonarQube. This is achieved by setting up a post-build action within the Jenkins pipeline. This action will evaluate the SonarQube analysis report and determine whether the code quality checks have been passed or failed.

A computer screen shot of a code

Description automatically generated

* 1. To further enhance the integration between SonarQube and Jenkins, we'll configure a webhook in SonarQube. This webhook will send notifications to our Jenkins server whenever a specific event occurs in SonarQube, such as the completion of an analysis. To set up the webhook, we'll navigate to the administration section of SonarQube, then to the configuration settings. Under the webhook configuration, we'll create a new webhook by providing a name and specifying the URL. This URL should point to a specific endpoint on our Jenkins server, typically incorporating the public IP address of the Jenkins server and a designated path, such as /sonarqube-webhook. After configuring the webhook, we can trigger a new build in Jenkins. Once the analysis is complete, we should be able to observe the latest delivery in the webhook's history on the SonarQube administration page.

A screenshot of a computer

Description automatically generated  
A screenshot of a webbook

Description automatically generated  
  
A close up of a computer screen

Description automatically generated

* 1. After checking the quality of the code, we can proceed to creating the AWS ECR private repositories. After creating the AWS ECR, proceed to steps to build the image. First step is to ensure that Jenkins is part of the docker group, as it will allows the jenkins user to execute Docker commands without requiring sudo privileges. Then, specify the environment by putting the AWS registry link and the region of the registry. Then, use the command shown in the image below in Jenkins’ pipeline. Build the pipeline and then go back to the build server and look at the created image.

A screenshot of a computer

Description automatically generated  
A black background with white text

Description automatically generated  
A close-up of a website

Description automatically generated  
A computer code with text

Description automatically generated with medium confidence  
A black screen with white text

Description automatically generated

* 1. After successfully building the Docker image, the next step is to push it to the Amazon Elastic Container Registry (ECR) repository. This will make the image accessible to other services and environments. To push the image, we'll execute a series of Docker commands. Once the push is complete, we can verify the successful deployment of the image by inspecting the ECR repository.

A close-up of a computer screen

Description automatically generated  
A screenshot of a computer

Description automatically generated

# Problem and Solutions

* 1. Problem 1:

Error when building the image in Jenkins:

Also:   org.jenkinsci.plugins.workflow.actions.ErrorAction$ErrorId: bd2a3437-25bb-4fc5-bfcd-944b3fb37107

groovy.lang.MissingPropertyException: No such property: docker for class: groovy.lang.Binding

Solution:

The Docker Pipeline plugin is missing. Install it int the plugin section.

A screenshot of a chat

Description automatically generated

* 1. Problem 2:

The error happened during the image building in Jenkins. The error specified on limited permission that hinders the building.

A close up of a computer code

Description automatically generated

Solution:

Put the Jenkins user to docker to allow it use docker without being root user. Restart both docker and Jenkins.

A close up of a computer code

Description automatically generated