# Project Overview

* 1. **Project Title:** Deploy an Application to a Kubernetes Cluster using Jenkins CICD
  2. **Summary:**

A diagram of a person

Description automatically generated  
  
**Setting Up the Infrastructure**

To establish a robust CI/CD pipeline, we'll set up three primary servers:

1. **Jenkins Server:** This server will act as the automation hub. It will be integrated with GitHub to automatically trigger builds and deployments whenever code changes are pushed to the repository.
2. **Ansible Server:** This server will serve as the deployment engine. It will use Ansible playbooks to automate the transfer of files, build Docker images, and deploy applications to the Kubernetes cluster.
3. **Kubernetes Cluster:** This cluster will host the deployed applications. It will be configured to receive and deploy the Docker images built by the Ansible server.

**Creating the GitHub Repository**

We'll create a centralized GitHub repository to store all project code, configuration files, and deployment scripts. This repository will serve as the single source of truth for the project.

**Configuring Jenkins**

1. **Webhook Integration:** We'll set up a webhook in GitHub to notify Jenkins whenever code changes are pushed to the repository.
2. **Job Creation:** We'll create a Jenkins job that will be triggered by the GitHub webhook.
3. **Pipeline Stages:** The Jenkins job will consist of the following stages:
   * **Checkout Code:** Clone the repository from GitHub.
   * **Transfer Files:** Transfer necessary files to the Ansible server.
   * **Trigger Ansible Playbook:** Initiate the Ansible playbook to build the Docker image and deploy the application to Kubernetes.

**Configuring Ansible**

1. **Playbook Creation:** We'll create Ansible playbooks to automate the following tasks:
   * **File Transfer:** Transfer files from the Jenkins server to the Ansible server and Kubernetes cluster.
   * **Docker Image Build:** Build the Docker image using the Dockerfile.
   * **Docker Image Push:** Push the built Docker image to the Docker Hub repository.
   * **Kubernetes Deployment:** Deploy the Docker image to the Kubernetes cluster using Kubernetes manifests.

**Deploying to Kubernetes**

The Ansible playbook will interact with the Kubernetes API to deploy the Docker image as a Kubernetes pod. The Kubernetes cluster will pull the Docker image from the Docker Hub repository and run it as a container.

* 1. **Project Flow:**

**1. Developer Pushes Code to GitHub**

* Developer makes changes to the code.
* Developer commits and pushes the changes to the GitHub repository.

**2. GitHub Webhook Triggers Jenkins Job**

* GitHub detects the code push and sends a notification to the Jenkins server via a webhook.

**3. Jenkins Job Starts**

* Jenkins receives the webhook notification and initiates a new build job.

**4. Jenkins Clones the Repository**

* Jenkins clones the latest version of the repository from GitHub to its workspace.

**5. Jenkins Transfers Files to Ansible Server**

* Jenkins transfers necessary files (e.g., Dockerfile, configuration files) to the Ansible server using SSH or other secure file transfer methods.

**6. Ansible Server Builds the Docker Image**

* Ansible executes a playbook to:
  + Retrieve the Dockerfile from the Ansible server.
  + Build the Docker image using the Docker build command.
  + Tag the Docker image with a unique identifier.

**7. Ansible Server Pushes the Docker Image to Docker Hub**

* Ansible executes a playbook to push the tagged Docker image to the Docker Hub repository.

**8. Ansible Server Deploys to Kubernetes**

* Ansible executes a playbook to:
  + Transfer deployment and service YAML files to the Kubernetes cluster.
  + Apply the YAML files to the Kubernetes API to create deployments and services.
  + Pull the Docker image from Docker Hub.
  + Deploy the Docker image as containers within the Kubernetes pods.

**9. Kubernetes Deploys the Application**

* Kubernetes API receives the deployment and service YAML files.
* Kubernetes scheduler schedules the pods to nodes in the cluster.
* Kubernetes container runtime pulls the Docker image from Docker Hub.
* Kubernetes container runtime starts the containers within the pods.

**10. Application is Live**

* The application is now accessible and operational in the Kubernetes cluster.
  1. **Purpose and Goals:**

The purpose of this project is to streamline the deployment and delivery of applications using a fully automated CI/CD pipeline. This setup integrates version control, continuous integration, configuration management, containerization, and orchestration tools to ensure seamless delivery of code changes from developers to production Kubernetes clusters.

The specific goals of the project are:

* 1. **Kubernetes Cluster for Orchestration**:
* Deploy application workloads on a Kubernetes cluster using Ansible playbooks.
* Ensure high availability and load balancing using Kubernetes services and deployment configurations.
  1. **End-to-End Automation**:
* Enable full automation of the CI/CD pipeline:
  + Detect changes in the GitHub repository.
  + Trigger Jenkins jobs for building, testing, and deploying applications.
  + Build, push, and deploy Docker images and Kubernetes configurations seamlessly.
  1. **Collaboration and Scalability**:
* Foster team collaboration by using GitHub for version control and Jenkins for workflow automation.
* Use Kubernetes for scaling applications as needed based on demand.
  1. **Prerequisites:**

# Implementation

* 1. To initiate our CI/CD pipeline, we'll begin by launching two AWS EC2 instances: one for Jenkins and another for Ansible. These instances will serve as the foundation for our automated deployment process. While specific configurations may vary based on project requirements, the primary goal is to establish two independent instances for these essential tools.

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated  
A screenshot of a software

Description automatically generated  
A screenshot of a computer error

Description automatically generated  
A screenshot of a computer

Description automatically generated  
A close-up of a computer screen

Description automatically generated  
A screenshot of a computer

Description automatically generated

* 1. Utilize MobaXterm to access the Jenkins server.

A screenshot of a computer

Description automatically generated  
A screenshot of a computer

Description automatically generated

* 1. Next, search for "Jenkins installation on Ubuntu" on a search engine. Click the first link from the official Jenkins website and select the Linux distribution, Ubuntu. Follow the provided steps for installing Java and the weekly Jenkins release. Once Jenkins and its dependencies are installed, configure the security group associated with the Jenkins server. Allow inbound traffic on port 8080 from any source to enable remote access.

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  
A computer screen with white text

Description automatically generated  
A black background with white text

Description automatically generated  
  
A screenshot of a computer

Description automatically generated

* 1. To access the Jenkins web interface, use the public IP address of the Jenkins server and port 8080. To unlock Jenkins, locate the provided path and execute the command "cat <given path>" on the Jenkins server. This will display a password. Copy and paste this password into the Jenkins interface and select the option to install suggested plugins.

A screenshot of a computer

Description automatically generated  
A black background with white text

Description automatically generated  
A screenshot of a computer

Description automatically generated  
A screenshot of a computer

Description automatically generated

* 1. To manage Jenkins, navigate to the plugin manager and search for the SSH Agent plugin. Install this plugin and then restart Jenkins to activate the changes.

A white rectangular object with a white background

Description automatically generated  
A screenshot of a computer

Description automatically generated  
A screenshot of a chat

Description automatically generated  
A close-up of a white background

Description automatically generated

* 1. Access the Ansible server using Mobaxterm, inputting the public IP address, username, and private key. Once connected, search for "Ansible installation on Ubuntu" and follow the official documentation's instructions. After installing Ansible, search for "Docker installation on Ubuntu" and follow the official documentation's steps to install Docker using the APT repository.

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 white background with blue and purple text

Description automatically generated  
A screenshot of a computer

Description automatically generated  
A computer screen with white text

Description automatically generated  
A screenshot of a web page

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. Next, create a new repository on GitHub and add a README file. On your local machine, create a new folder, open Git Bash within that folder, and clone the newly created repository. Switch to the main branch of the project and ensure your Git username and email address are correctly configured.

A screenshot of a computer

Description automatically generated  
A screenshot of a computer

Description automatically generated  
A computer screen with text on it

Description automatically generated  
A screen shot of a computer

Description automatically generated  
A screen shot of a computer

Description automatically generated  
A black background with yellow text

Description automatically generated

* 1. In Git Bash, create a Dockerfile, which serves as a blueprint for the Docker image. This Dockerfile outlines the base operating system, required package installations, website download (in this case, from the internet), port exposure, and other configurations. Once the Dockerfile is created, add, commit, and push it to the GitHub repository.

A computer screen shot of a computer code

Description automatically generated  
A screenshot of a computer screen

Description automatically generated  
A black screen with yellow text

Description automatically generated  
A screen shot of a computer

Description automatically generated  
A screenshot of a computer screen

Description automatically generated

* 1. Return to the Jenkins dashboard and create a new pipeline item. The initial stage of this pipeline will focus on checking out or cloning the GitHub repository into the Jenkins workspace. Utilize pipeline syntax to automate the necessary commands. Apply and save the pipeline configuration. Trigger a build and verify that the pipeline executes successfully. Access the Jenkins server and navigate to the /var/lib/Jenkins/workspace directory to view the cloned repository.

A screenshot of a computer

Description automatically generated  
A screenshot of a computer

Description automatically generated  
A computer screen shot of a computer code

Description automatically generated  
A screenshot of a computer

Description automatically generated  
A white background with a black border

Description automatically generated with medium confidence  
A computer screen shot of a computer

Description automatically generated

* 1. To automate our workflow, we'll establish a GitHub webhook to trigger Jenkins jobs whenever a file in our GitHub repository is modified or added. In Jenkins, enable the "GitHub hook trigger for GITScm polling" option for our pipeline. Next, navigate to GitHub's settings and add a new webhook. Here, we'll input the public IP address of our Jenkins server along with its port as the payload URL, ensuring it's configured to accept JSON payloads. To enhance security, we'll generate an API token in Jenkins and input it as the secret in the GitHub webhook form. By completing these steps, we'll successfully integrate GitHub and Jenkins, automating the build and deployment process.

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 webbook

Description automatically generated  
A screenshot of a chat

Description automatically generated  
A screenshot of a webpage

Description automatically generated

* 1. To facilitate the Docker image build process, we'll establish SSH access from our Jenkins server to the Ansible server. This will allow us to securely transfer the Dockerfile from Jenkins to the Ansible server, where the image will be constructed. Within the Jenkins pipeline syntax, we'll configure an SSH agent using a private key to authenticate with the Ansible server. This involves generating credentials for the SSH connection and integrating them into the pipeline. The initial pipeline stage will initialize the SSH connection to the Ansible server, specifying the username and IP address. Subsequently, the Dockerfile will be copied from its directory on the Jenkins server to the /home/ubuntu directory on the Ansible server, preparing it for the image build process.

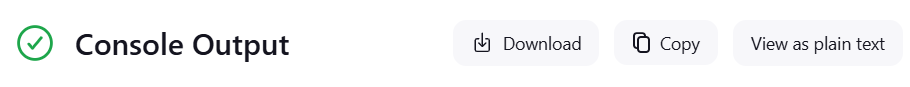
A screenshot of a computer

Description automatically generated  
A computer screen shot of a computer code

Description automatically generated  


* 1. The subsequent stage in the Jenkins pipeline, executed on the Ansible server, is dedicated to building the Docker image. This involves executing a Docker build command, specifying the Dockerfile's location as the build context. Upon successful completion of the build process, the newly created Docker image will be accessible on the Ansible server, ready for deployment.

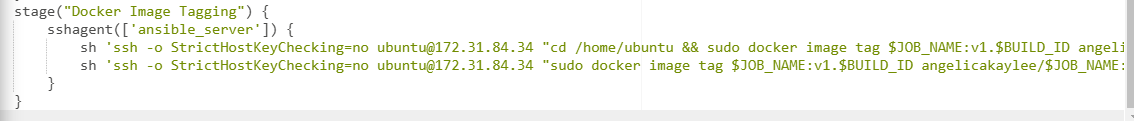
A screenshot of a computer program

Description automatically generated  
  
A screenshot of a computer

Description automatically generated

* 1. To facilitate image storage and distribution, we'll create a Docker Hub account and establish a repository for our project's images. Returning to the Jenkins pipeline, we'll tag the newly built Docker image with a combination of the Docker Hub account name, repository name, version number, and build number. Additionally, we'll create a "latest" tag for the most recent build. This tagging strategy enables us to manage multiple image versions and easily deploy the desired one.

A screenshot of a computer

Description automatically generated  
  
  
A screen shot of a computer

Description automatically generated

* 1. To push the Docker images to Docker Hub, we'll utilize the withCredentials step in the Jenkins pipeline. This step secures our Docker Hub credentials by storing them as a secret text. By configuring the withCredentials block with the appropriate credentials, we'll grant the Jenkins pipeline the necessary permissions to authenticate with Docker Hub and push the tagged images to the designated repository.

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  
A white rectangular object with a black stripe

Description automatically generated  
A screenshot of a web page

Description automatically generated

* 1. To deploy our Docker image to a Kubernetes cluster, we'll initiate a new Ubuntu server instance on a cloud platform, opting for a t2.medium instance type. Once the server is up and running, we'll leverage MobaXterm, utilizing the public IP address, username, and private key file to establish an SSH connection to the Kubernetes server. This connection will provide us with remote access to the server, allowing us to proceed with the Kubernetes cluster setup and deployment process.

A screenshot of a computer

Description automatically generated  
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

Description automatically generated

* 1. To set up a local Kubernetes cluster on the Ubuntu server, we'll follow a series of steps. Initially, we'll install the necessary prerequisites, including Docker and kubectl. However, due to compatibility issues with older versions of minikube, we'll opt for a different approach. To ensure proper operation, we'll add the Ubuntu user to the Docker group, as root users cannot directly run the minikube server. Finally, we'll start the minikube server as a Docker container, effectively creating a local Kubernetes environment on our Ubuntu server.

A screenshot of a computer

Description automatically generated  
  
A screen shot of a computer

Description automatically generated

* 1. To enable passwordless access from the Ansible server to the Kubernetes server, start by editing the /etc/ssh/sshd\_config file on the Kubernetes server. Set PermitRootLogin to yes and PasswordAuthentication to yes to allow secure SSH access. On the Ansible server, generate an RSA key pair by running ssh-keygen -t rsa, which creates id\_rsa (private key) and id\_rsa.pub (public key) files. Typically, you would use the ssh-copy-id ubuntu@<private\_ip\_address> command to transfer the public key to the Kubernetes server, enabling key-based authentication. However, if this approach encounters issues despite the SSH configuration changes, a manual method can be employed. Copy the contents of the id\_rsa.pub file on the Ansible server and append them to the ~/.ssh/authorized\_keys file on the Kubernetes server. Finally, verify the connection by running ssh ubuntu@<private\_ip\_address> from the Ansible server to ensure seamless, passwordless access.

A computer screen shot of a computer program

Description automatically generated  
  
  
  
A computer screen shot of a computer screen

Description automatically generated  
A screenshot of a computer

Description automatically generated  
A computer screen shot of a computer program

Description automatically generated  
A screenshot of a computer

Description automatically generated  
  
A computer screen with text and numbers

Description automatically generated

* 1. To initiate Ansible's management of the Kubernetes server, we'll first configure the hosts file. Create a new group, such as "node" or "Kubernetes," and list the private IP address of the Kubernetes server within this group. This will enable Ansible to recognize and target the server. To verify the connection, execute the command ansible -m ping node. A successful execution of this command confirms Ansible's ability to communicate with the specified node.

A screenshot of a computer

Description automatically generated  
A computer screen shot of a computer

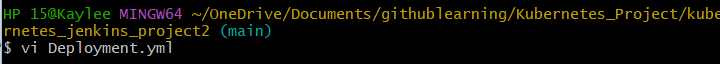
Description automatically generated  
A screenshot of a computer

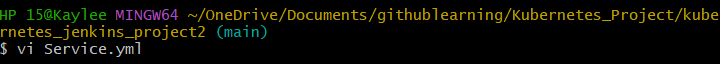
Description automatically generated  
A computer screen shot of a computer

Description automatically generated  
A computer screen shot of a computer screen

Description automatically generated

* 1. Now, let's transition back to your local machine's Git Bash. Here, we'll create the necessary files for our deployment: a Deployment file, a Service file, and an Ansible playbook. Once these files are ready, push them to your GitHub repository. This will ensure that they are accessible and version-controlled.

  
A screenshot of a computer

Description automatically generated  
  
A screen shot of a computer

Description automatically generated  
  
A screenshot of a computer program

Description automatically generated

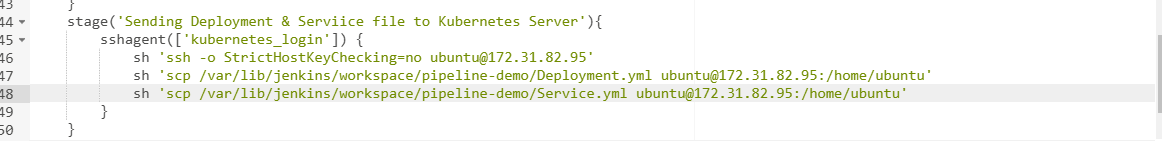
* 1. To proceed with the Jenkins pipeline, return to the pipeline syntax and configure the credentials for the Kubernetes server using SSH Agent. This will allow Jenkins to securely connect to the Kubernetes server and execute the necessary tasks.

A screenshot of a computer

Description automatically generated  
A screenshot of a computer

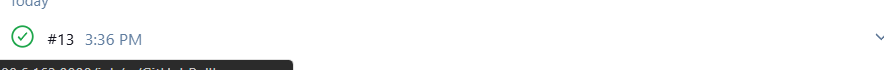
Description automatically generated

* 1. Leveraging the SSH Agent on the Jenkins server, we'll securely log into the Kubernetes server. Once connected, the Deployment and Service files will be transferred to the /home/ubuntu directory on the Kubernetes server. To validate the pipeline's success, we can verify the presence of these files on the Kubernetes server.

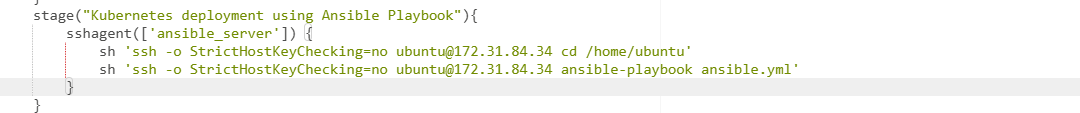
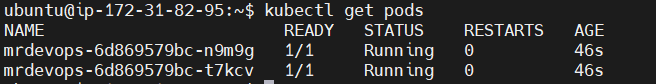
  
  
A screenshot of a calendar

Description automatically generated

* 1. The subsequent step in the pipeline involves transferring the Ansible playbook file to the Ansible server. This file will contain the automation instructions required to deploy the Kubernetes resources.


* 1. The pipeline will then proceed to access the Ansible server and execute the Ansible playbook. This playbook will establish a secure, passwordless SSH connection to the Kubernetes server and subsequently deploy the specified Deployment and Service files.

* 1. Change the inbound rules to allow access to the website.

A screenshot of a computer

Description automatically generated

# Problem and Solutions

* 1. Problem:

The problem arose because the kubectl command was being run with elevated privileges (root) due to the become: true directive in the Ansible playbook. This caused the Kubernetes configuration file (~/.kube/config), which is typically stored in the home directory of the ubuntu user, to become inaccessible since the command was executed as root.

**Summary of the Problem:**

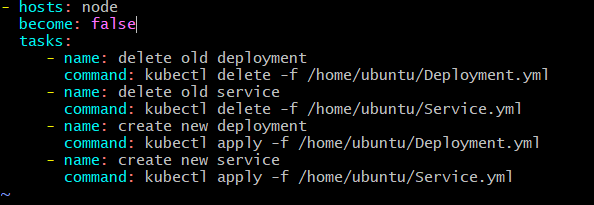
* **Root Cause:** The kubectl command could not locate the correct Kubernetes configuration file because it was run with elevated privileges, causing it to default to an incorrect or nonexistent configuration.

A close up of a text

Description automatically generated

Solution:

In the ansible file make the “become” command into false to remove the need to elevate privilege to root. Thus, the commands will be ran as ubuntu user.



Explanation:

**The Key Points:**

1. **Multiple .minikube Directories**:
   * There are two .minikube directories:
     + **/home/ubuntu/.minikube**: Used when running commands as the ubuntu user.
     + **/root/.minikube**: Used when running commands as root (or with elevated privileges like become: true).
2. **Impact of become: true**:
   * When become: true is set in your Ansible playbook, tasks are executed as the root user.
   * Minikube, when run as root, attempts to access /root/.minikube instead of /home/ubuntu/.minikube.
   * Since /root/.minikube doesn’t contain the necessary configuration files (like the certificates, kubeconfig, etc.) from the Minikube setup for ubuntu, commands fail.
3. **Missing SSH Key**:
   * If passwordless SSH is configured for the ubuntu user, the public key will reside in /home/ubuntu/.ssh/authorized\_keys.
   * When Ansible elevates to root, it looks for the key in /root/.ssh/authorized\_keys, which does not contain the necessary credentials for passwordless SSH.
4. **How It Leads to Errors**:
   * **File and Directory Access**: Commands executed as root interact with /root/.minikube, which lacks the expected files and configurations.
   * **Lock Errors**: The root user tries to acquire locks or access files in /root/.minikube but doesn’t find them.
   * **Inconsistent Environment**: Minikube and Kubernetes components (like kubectl) expect consistent configuration paths, and switching user contexts breaks this assumption.

**Why Does It Work When become: false?**

When become: false, all tasks are executed as the ubuntu user:

* Minikube accesses /home/ubuntu/.minikube, which has the correct configuration files.
* SSH key access remains consistent because Ansible uses the ubuntu user, matching the Minikube setup.
* There is no context switch to root, avoiding file ownership and path inconsistencies.

**Lessons Learned**

1. **User-Specific Configuration**:
   * Minikube setup is user-specific, and switching contexts (e.g., ubuntu to root) can break the configuration.
2. **Avoid become: true Unless Necessary**:
   * Use become: true sparingly and only for tasks that genuinely require root privileges (e.g., package installation or system-wide configuration).
3. **Ensure Consistency in User Context**:
   * If you intend to run everything as the ubuntu user, ensure all related tools and configurations (e.g., .minikube, .ssh) are in /home/ubuntu.