TESTS AUTOMATION WITH ROBOT FRAMEWORK AND DEVOPS INTEGRATION

Introduction

In this project, we integrate test automation with Robot Framework into a DevOps pipeline. The setup utilizes Jenkins hosted on Kubernetes, with Kubernetes acting as the agent and GitLab serving as the remote repository.

Selenium and Robot Framework Setup to run tests on a docker container

1. **Python Script: chrome_options.py**

This script is used to configure Chrome browser options for headless operation, sandboxing, and shared memory usage.

```
from selenium import webdriver

from selenium.webdriver.chrome.service import Service

from selenium.webdriver.chrome.options import Options


def get_chrome_options():

    chrome_options = Options()

    chrome_options.add_argument("--headless")

    chrome_options.add_argument("--no-sandbox")

    chrome_options.add_argument("--disable-dev-shm-usage")

    return chrome_options
```

2. **Robot Framework Resource File: `resource.robot`**

This file defines the reusable keywords and variables for the test cases, incorporating the custom Chrome options for headless operation.

```
*** Settings ***

Documentation     A resource file with reusable keywords and variables.

Library           SeleniumLibrary

Library           OperatingSystem
```

```
Library           Collections

Library           ../chrome_options.py    # Import the custom options
script


*** Variables ***

${SERVER}         localhost:7272

${BROWSER}        chrome

${DELAY}          0

${VALID_USER}     demo

${VALID_PASSWORD}    mode

${LOGIN_URL}      http://${SERVER}/

${WELCOME_URL}    http://${SERVER}/welcome.html

${ERROR_URL}      http://${SERVER}/error.html

${CHROME_OPTIONS}  --headless;--no-sandbox;--disable-dev-shm-usage



*** Keywords ***

Open Browser To Login Page

    ${chrome_options} =  Evaluate
sys.modules['selenium.webdriver'].ChromeOptions()  sys,
selenium.webdriver

    Call Method  ${chrome_options}  add_argument  --headless

    Call Method  ${chrome_options}  add_argument  --no-sandbox

    Call Method  ${chrome_options}  add_argument
--disable-dev-shm-usage

    Open Browser    ${LOGIN_URL}    ${BROWSER}
options=${chrome_options}

    Maximize Browser Window

    Set Selenium Speed    ${DELAY}

    Login Page Should Be Open
```

Jenkins Deployment in Kubernetes

This guide outlines the steps to deploy Jenkins in a Kubernetes cluster using the provided YAML configuration files. The configurations include a PersistentVolumeClaim, Service Account, Role, Role Binding, Deployment, and Services. These files also contain configuration to run kubernetes pod as a jenkins controller to execute pipeline jobs.

Prerequisites

- A running Kubernetes cluster.
- `kubectl` command-line tool configured to interact with the cluster.
- A default StorageClass in the Kubernetes cluster.

Configuration Files

1. Create Namespace ‘ops’

This configuration creates the `ops` namespace.

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: ops
```

2. PersistentVolumeClaim

This configuration creates a PersistentVolumeClaim (PVC) for Jenkins data storage.

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: jenkins-pvc
  namespace: ops
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: standard #default storage class
  resources:
    requests:
      storage: 10Gi
```

3. Service Account

This configuration creates a Service Account for Jenkins with administrative privileges within the `ops` namespace.

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: jenkins-admin
  namespace: ops
```

4. Role

This configuration creates a Role that grants permissions to manage pods within the `ops` namespace.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: pod-manager
  namespace: ops
rules:
- apiGroups: [""]
  resources: ["pods", "pods/log", "pods/exec"]
  verbs: ["create", "delete", "get", "list", "patch", "update",
"watch"]
```

5. Role Binding

This configuration binds the `pod-manager` Role to the `jenkins-admin` Service Account within the `ops` namespace.

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: pod-manager-binding
  namespace: ops
subjects:
- kind: ServiceAccount
  name: default
  namespace: ops
roleRef:
  kind: Role
  name: pod-manager
  apiGroup: rbac.authorization.k8s.io
```

6. Deployment

This configuration defines the Jenkins deployment.

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: jenkins
  namespace: ops
spec:
  replicas: 1
  selector:
    matchLabels:
      app: jenkins
  template:
    metadata:
      labels:
```

```
        app: jenkins
    spec:
      securityContext:
        fsGroup: 1000
        runAsUser: 1000
      containers:
      - image: jenkins/jenkins:lts
        imagePullPolicy: IfNotPresent
        name: jenkins
        ports:
        - containerPort: 8080
          name: http-port
        - containerPort: 50000
          name: jnlp-port
        volumeMounts:
        - mountPath: /var/jenkins_home
          name: jenkins-data
      volumes:
        - name: jenkins-data
          persistentVolumeClaim:
            claimName: jenkins-pvc
```

7. Services

This configuration defines the services for Jenkins.

HTTP Service

This service exposes Jenkins on port 8080 using a NodePort.

```
apiVersion: v1
kind: Service
```

```
metadata:
  name: jenkins
  namespace: ops
spec:
  type: NodePort
  ports:
    - port: 8080
      targetPort: 8080
      nodePort: 30000
  selector:
    app: jenkins
```

JNLP Service

This service exposes Jenkins JNLP port 50000 using a ClusterIP.

```
apiVersion: v1
kind: Service
metadata:
  name: jenkins-jnlp
  namespace: ops
spec:
  type: ClusterIP
  ports:
    - port: 50000
      targetPort: 50000
  selector:
    app: jenkins
```

8. Apply all settings

run “kubectl apply -f file” and replace file with names of your files one by one.

Access and configure jenkins

To look for jenkins url. Run in terminal

```
kubectl get nodes -o wide
```

and take the node INTERNAL-IP as <node-ip>
Now, when browsing to any one of the Node IPs on port 30000, you will be able to access the Jenkins dashboard.

```
http://<node-ip>:30000
```

Jenkins will ask for the initial Admin password when you access the dashboard for the first time.

You can get that from the pod logs either from the Kubernetes dashboard or CLI. You can get the pod details using the following CLI command.

```
kubectl get pods --namespace=ops
```

With the pod name, you can get the logs as shown below. Replace the pod name with your pod name.

```
kubectl logs <jenkins-pod-name> --namespace=ops
```

The password can be found at the end of the log.

Alternatively, you can run the exec command to get the password directly from the location as shown below.

```
kubectl exec -it <jenkins-pod-name> cat /var/jenkins_home/secrets/initialAdminPassword -n ops
```

Once you enter the password, proceed to install the suggested plugin and create an admin user. All of these steps are self-explanatory from the Jenkins dashboard.

```
install suggested plugins
```

then, you can continue as administrator or create a user.
After that, you need to install necessary plugins through **Manage Jenkins > Plugins** under available plugins.

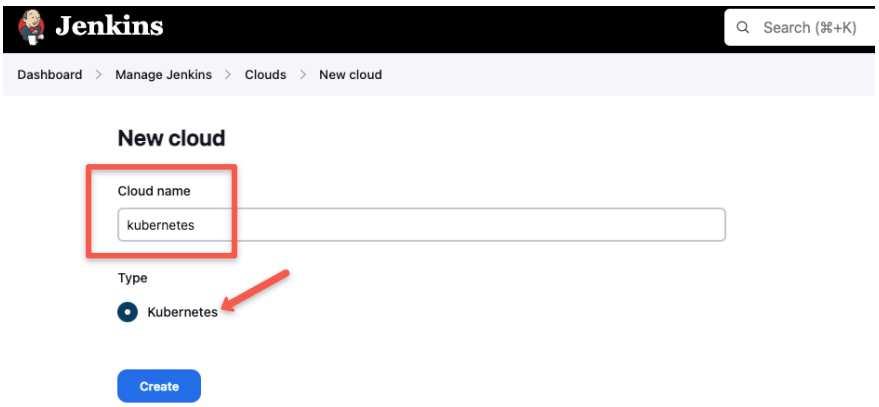
- Gitlab
- Docker
- Kubernetes

Create a Kubernetes Cloud Configuration

go to `Manage Jenkins` –> `Clouds`

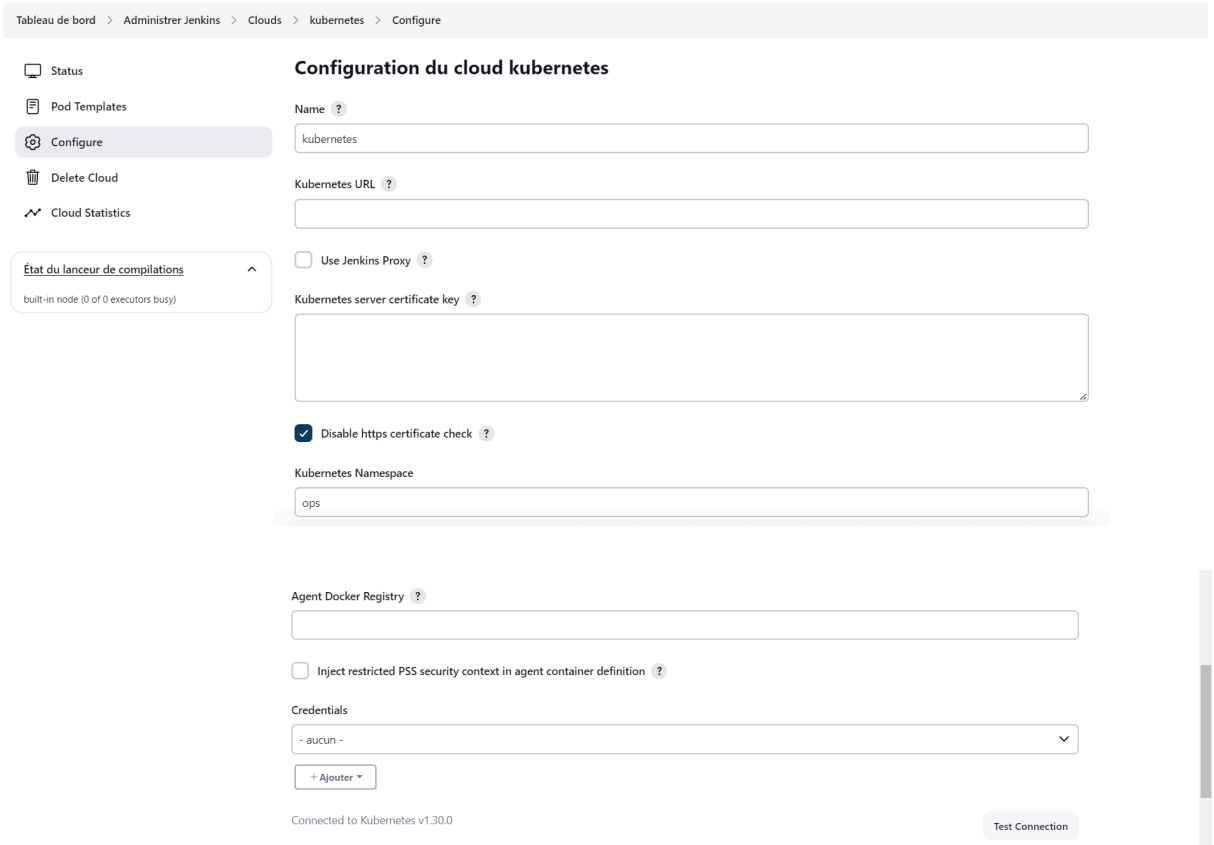
Click New Cloud.

Give a name and select Kubernetes.



Configure Jenkins Kubernetes Cloud

Since we have Jenkins inside the Kubernetes cluster with a service account to deploy the agent pods, we don't have to mention the Kubernetes URL or certificate key.
However, to validate the connection using the service account, use the Test Connection button as shown below. It should show a connected message if the Jenkins pod can connect to the Kubernetes API Server.



Configure the Jenkins URL Details

The jenkins pod exposed by a Service have the following DNS resolution available:

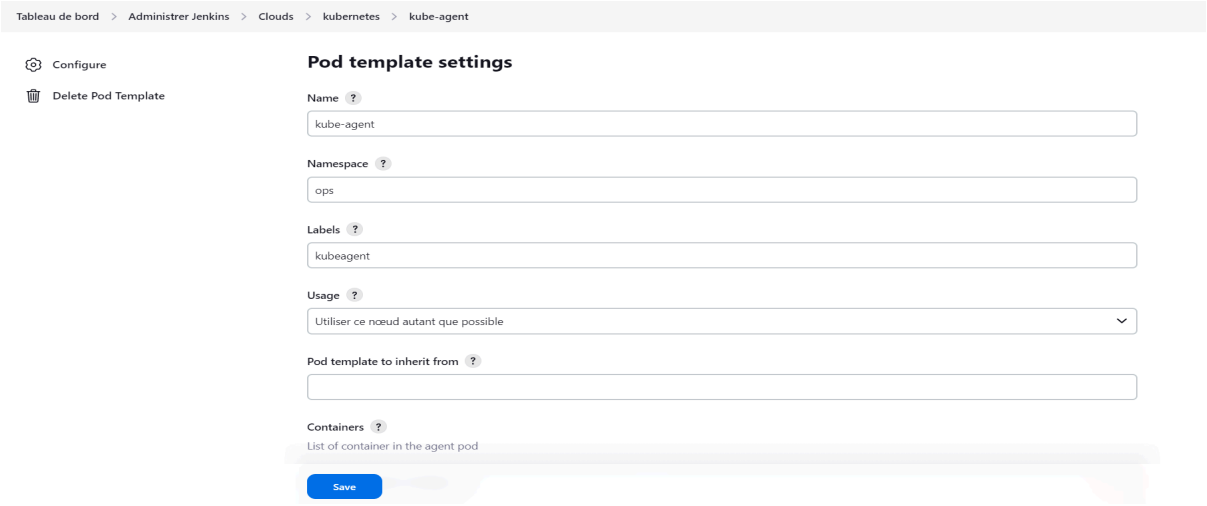
`http://jenkins_pod_name.service_name.my_namespace.svc.cluster-domain.example:service_port`

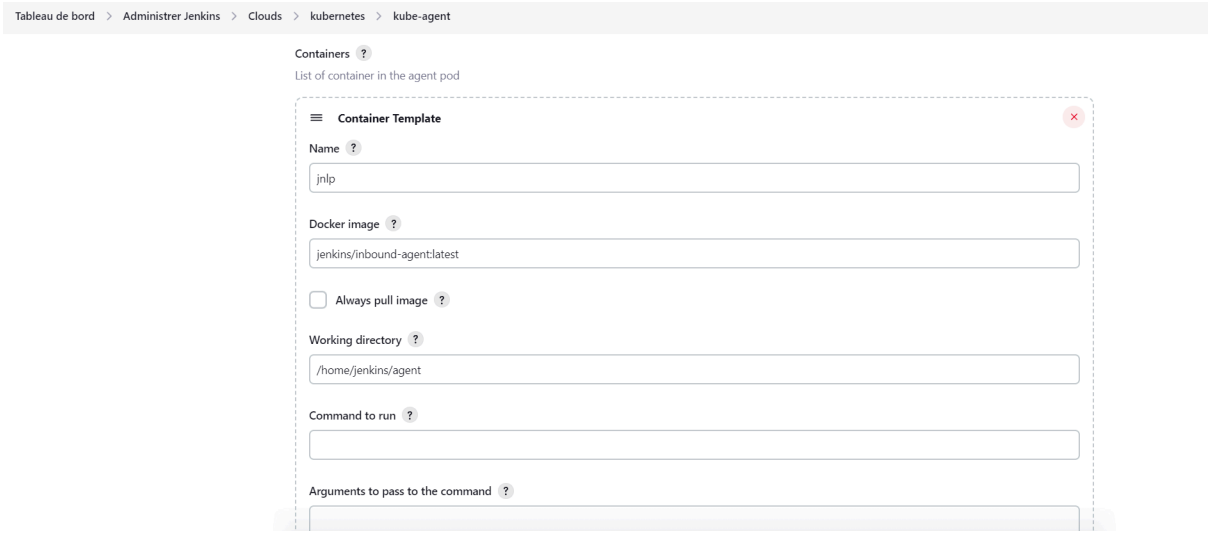
in our case

`http://jenkins-0.jenkins.ops.svc.cluster.local:8080`

Also, add the POD label, which can be used to group the containers that can be used for billing or custom build dashboards.

Next, you must add the POD template with the details, as shown in the image below. The label kube-agent will be used as an identifier to pick this pod as the build agent. Next, we must add a container template with the Docker image details.





Ensure that you remove the `sleep` and `9999999` default argument from the container template.

Configure Service Account to modify deployment in Myapp namespace

```
apiVersion: rbac.authorization.k8s.io/v1
```

```
kind: Role

metadata:

  namespace: myapp  # Target namespace

  name: robot-modify-access

rules:

- apiGroups: ["apps"]  # For Deployments

  resources: ["deployments"]

  verbs: ["get", "list", "watch", "create", "update", "patch",
"delete"]
```

```
- apiGroups: [""]

  resources: ["services"]

  verbs: ["get", "list", "watch", "create", "update", "patch",
"delete"]

- apiGroups: [""]  # Core API group

  resources: ["pods", "pods/log", "pods/exec"]

  verbs: ["create", "delete", "get", "list", "patch", "set", "update",
"watch"]




---



apiVersion: rbac.authorization.k8s.io/v1

kind: RoleBinding

metadata:

  name: ops-serviceaccount-modify-robot

  namespace: myapp  # Target namespace

roleRef:

  apiGroup: rbac.authorization.k8s.io

  kind: Role
```

```
  name: robot-modify-access  # This refers to the Role in the spring
namespace


subjects:


- kind: ServiceAccount


  name: jenkins-admin  # Name of the ServiceAccount


  namespace: ops  # Namespace of the ServiceAccount
```

Configure GitLab in Jenkins

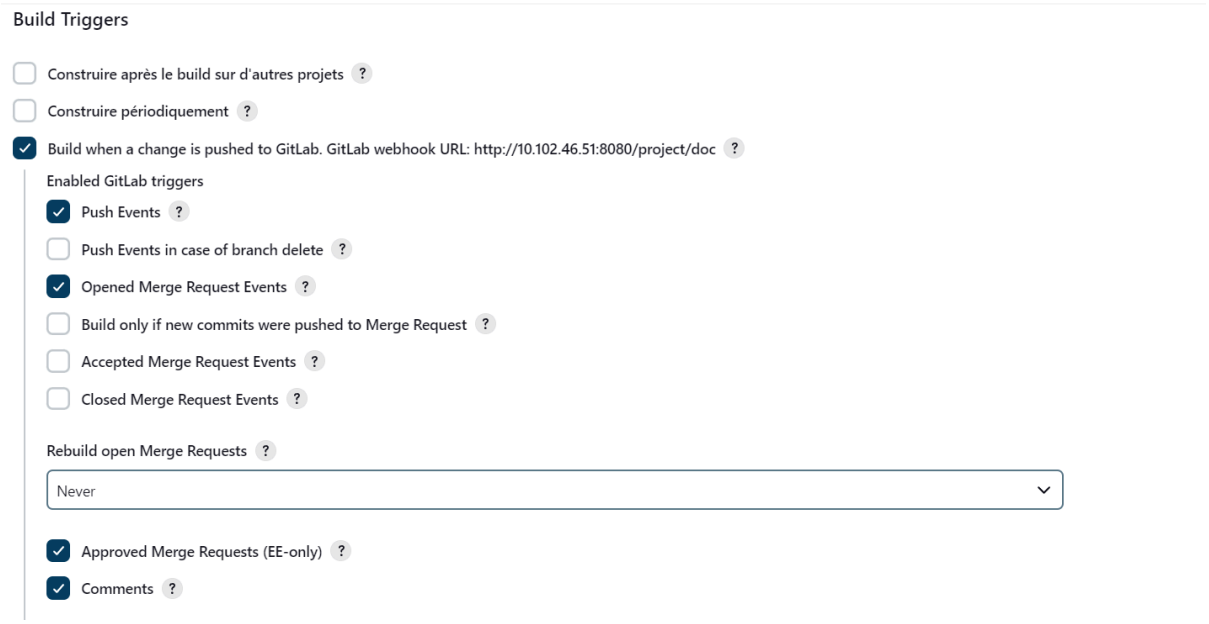
1. Select Manage Jenkins > Configure System.
2. In the GitLab section, select Enable authentication for '/project' end-point.
3. Select Add, then choose Jenkins Credential Provider.
4. Select GitLab API token as the token type.
5. In API Token, paste the access token value you copied from GitLab and select Add.
6. Enter the GitLab server's URL in GitLab host URL.
7. To test the connection, select Test Connection.

Create the Project

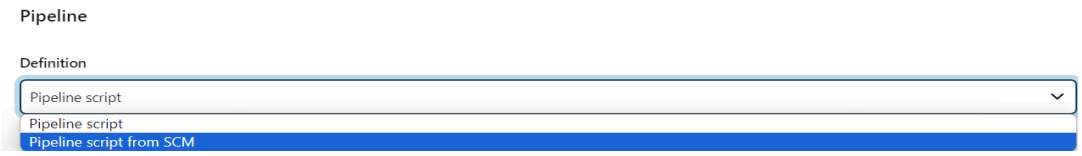
Go to Jenkins home –> New Item and create a pipeline project.

In this field specify the connection name previously created

then select Build when a change is pushed to GitLab to configure the webhook.



And set the pipeline definition to use Jenkinsfile.



Configure the Jenkins integration in gitlab

In this step it needed to have jenkins hosted and accessible from the internet (VPS or Cloud service) or have a gitlab server in your kubernetes cluster.

You can do this using this link:
https://docs.gitlab.com/ee/integration/jenkins.html#with-a-jenkins-server-url

Prepare the Dockerfile of the project

in this step you need to create the dockerfile that will be pushed to Dockerhub to deploy the application.

Define the Pipeline

Replace the white highlighted text with your own details

```
pipeline {

  agent {

    kubernetes {

      yaml '''
```

```
apiVersion: v1
kind: Pod
spec:
  resources:
    requests:
      memory: 50Mi
      cpu: 50m
  priorityClassName: high-priority
  serviceAccountName: jenkins-admin
  containers:
  - name: robot
    image: rapidfort/python-chromedriver
    command:
    - cat
    tty: true
  - name: docker
    image: docker:latest
    command:
    - cat
    tty: true
    volumeMounts:
```

```
            - mountPath: /var/run/docker.sock
              name: docker-sock
        - name: kubectl
          image: bitnami/kubectl:latest
          securityContext:
            runAsUser: 1000
          command:
          - "sleep"
          args:
          - "99d"
          tty: true
        volumes:
        - name: docker-sock
          hostPath:
            path: /var/run/docker.sock
      '''
  }
}
stages {
  stage('Clone') {
    steps {
```

```
        container('robot') {
          git branch: 'main', credentialsId: 'git-cred', url:
'https://gitlab.com/youssefmasmoudi16/robot_tests.git'
        }
      }
    }
    stage('robot-tests') {
      steps {
        container('robot') {
          sh 'pip install -r requirements.txt'
          sh 'python demoapp/server.py &'
          sh 'robot --outputdir ./reports login_tests'
        }
      }
      post {
        always {
          archiveArtifacts artifacts: 'reports/*', followSymlinks:
false
        }
      }
    }
    stage('Build-Docker-Image') {
```

```
      steps {

        container('docker') {

          sh 'docker build -t youssefmasmoudi/myserver:latest .'

        }

      }

    }

    stage('Login-Docker') {

      steps {

        container('docker') {

          withCredentials([usernamePassword(credentialsId:
'dockerhub-cred', passwordVariable: 'password', usernameVariable:
'user')]) {

            sh 'docker login -u ${user} -p ${password}'

           }

        }

      }

    }

    stage('Push-Images-Docker-to-DockerHub') {

      steps {

        container('docker') {

          sh 'docker push youssefmasmoudi/myserver:latest'

        }
```

```
      }

    }

    stage('Update Kubernetes Deployment') {

      steps {

        container('kubectl') {

          sh 'kubectl set image deployment/myserver-deployment
myserver=youssefmasmoudi/myserver:latest -n myapp'

        }

      }

    }

  }



  post {

    always {

      container('docker') {

        sh 'docker logout'

      }

    }

  }

}
```

Conclusion

This Jenkins pipeline is designed to run multiple jobs across different projects by leveraging containerized environments tailored to specific project needs. By using Kubernetes agents and defining custom containers for each stage, this pipeline ensures that the necessary dependencies and tools are isolated within their respective containers. This approach provides flexibility and scalability, making it easy to adapt the pipeline for various projects with different requirements. The use of containers like robot for testing and docker for building and deploying images ensures that each job runs in a consistent and controlled environment, minimizing conflicts and maximizing efficiency.