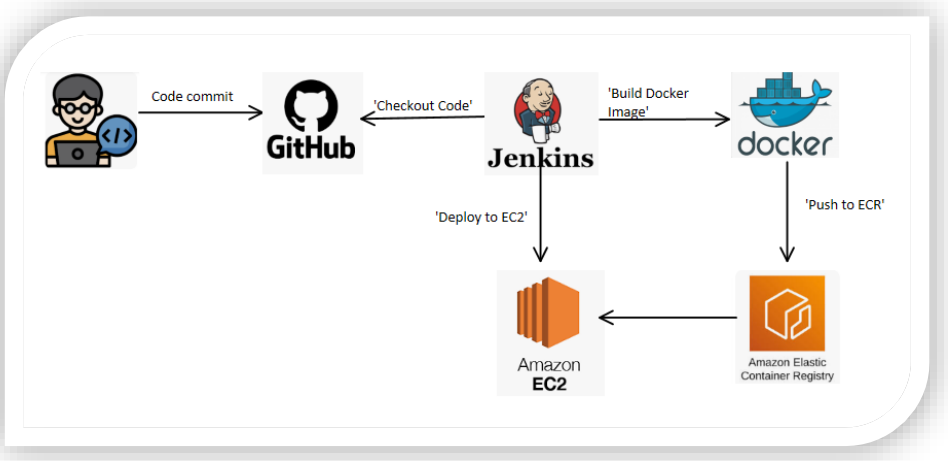
PROJECT 1: CI / CD Pipeline using Jenkins and deploy the real world Web Application

Project Description:
Build CI / CD Pipeline using Jenkins and deploy the real world Web Application in AWS Cloud

Goals:
CI/CD Pipelines will help you learn
Server automation, continuous integration, building pipelines, configuration of tools, automated testing, code quality improvement, and distributed systems in Jenkins through intensive, hands-on practical assignments.

Technologies Used:-
1. Jenkins
2. Groovy
3. AWS Cloud
4. Git
5. Docker



Steps:
1. Create jenkins file using our in-house code repo [should be cloned from git/bitbucket]
2. Create Docker file in the same repository
3. Build-Docker image with tagging as build version, unit test cases should pass if any for the code
4. The Image should be available in ECR with build version as TAG
5. The Docker Image should be deployed to EC2 Machine
6. The EC2 Machine Need to open specific Inbound Port and restrict Access only for admin user to login
7. Jenkins Jobs should do validation and display successful message
8. Report should be sent to e-mail and it should contain status of each JOB
9. Domain should be registered with AWS

Building a CI/CD Pipeline with Example

Implementation:

Prerequisites:

- An AWS account (Free tier can be used for initial setup)
- A Git repository (e.g., GitHub) for your code
- Docker installed on your machine
- Basic understanding of Python scripting

1. Setting Up Jenkins:

- Follow the official guide to install Jenkins on a server (https://www.jenkins.io/doc/book/installing/). There are also managed Jenkins services available.
- Java: `sudo dnf install java-17-amazon-corretto -y`
- Docker :
  - ➢ `sudo yum install docker -y`
  - ➢ `sudo usermod -aG docker jenkins`
  - ➢ `sudo systemctl restart jenkins`
- Install required plugins:
  - Search for and install plugins like "Git", "Docker", "Docker pipeline", "AWS SDK," and "Email Extension."

2. Creating the Dockerfile:

- Create a file named Dockerfile in the same directory as your Jenkinsfile.

Example Dockerfile (Simple Python App):
Dockerfile

```
FROM python:3.8-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install -r requirements.txt

COPY . .

CMD [ "python", "app.py" ]
```

Explanation:

- This Dockerfile defines instructions to build a Docker image based on a Python 3.8 slim image.
- The WORKDIR sets the working directory inside the container.
- It copies the requirements.txt file (containing Python dependencies) and installs them using pip.
- Then, it copies all files from the current directory (.) into the container.
- Finally, the CMD instruction defines the command to run when the container starts (in this case, it executes your Python application named app.py).

3. Simple Python Web App (app.py):

Create a basic Python script named app.py to demonstrate the deployment process.

Python

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello from a deployed web app!'

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

Explanation:

- This simple Flask application defines a route (/) that returns the message "Hello from a deployed web app!".
- You can replace this with your actual web application code.

4. Creating the Jenkinsfile:

- Clone your Git repository (Using GitHub).
- Create a new file named Jenkinsfile inside the repository.

5. Building and Pushing Docker Image to ECR with Tag Versions

The Jenkinsfile for building and pushing the Docker image to ECR, including version tags:

```
pipeline {
    agent any
    environment {
        AWS_ACCOUNT_ID="992382522294"
        AWS_DEFAULT_REGION= "ap-south-1"
        IMAGE_REPO_NAME="ecr_docker_images"
        IMAGE_TAG="IMAGE_TAG"
        REPOSITORY_URI="${AWS_ACCOUNT_ID}.dkr.ecr.${AWS_DEFAULT_REGION}.amazon
aws.com/${IMAGE_REPO_NAME}"
    }
```

```
    stages {
        stage('Login into AWS ECR'){
            steps{
                script{
                    sh "aws ecr get-login-password --region ${AWS_DEFAULT_REGION} | docker
login --username AWS --password-stdin
${AWS_ACCOUNT_ID}.dkr.ecr.${AWS_DEFAULT_REGION}.amazonaws.com"
                }
            }
        }
        stage('Checkout Code') {
            steps {
                checkout scmGit(branches: [[name: '*/main']], extensions: [],
userRemoteConfigs: [[url: 'https://github.com/Abdul-Quader/TestJen.git']])
            }
        }

        stage('Build Docker Image') {
            steps {
                sh 'docker build -t my-web-app:${BUILD_NUMBER} .' // Build with build number
tag
            }
        }

        stage('Push to ECR') {
            steps {
                script {
                    // Retrieve build version from environment variable
                    def buildVersion = "${BUILD_NUMBER}"
                    // Configure AWS credentials and ECR details here
                    sh 'aws ecr get-login-password --region ${AWS_DEFAULT_REGION} | docker
login --username AWS --password-stdin <REPOSITORY_URI>'
```

```
                // Tag image with build version and ECR repository URI
                sh "docker tag my-web-app:${buildVersion}
<REPOSITORY_URI>:${buildVersion}"
                // Push image to ECR with build version tag
                sh "docker push <REPOSITORY_URI>:${buildVersion}"
            }
        }
    }
    // ... other stages for deployment, etc.
  }
}
```

Explanation:

- The "environment" block declares environment variables, that would be used throughout the pipeline.
- The pipeline block defines the overall pipeline structure.
- The agent any specifies any available Jenkins agent can run the job.
- The stages block defines different stages in the pipeline:
  - Checkout Code: Fetches code from your Git repository using the git step.
- In the "Build Docker Image" stage, the docker build command now uses ${BUILD_NUMBER} as part of the image tag. This injects the current Jenkins build number into the tag, creating a unique version identifier for each build.
- The "Push to ECR" stage retrieves the build version from the ${BUILD_NUMBER} environment variable within a script block.
- The image is then tagged twice:
  - With the build number (my-web-app:${buildVersion}) for internal tracking.
  - With the ECR repository URI and build number (<REPOSITORY_URI>:<buildVersion>) for pushing to the ECR repository.
- And, the script pushes the image with the ECR-specific tag to your ECR repository.

6. **Deploying to EC2**

Updated Jenkinsfile demonstrating deployment to EC2 and post-build actions.

```
pipeline {
  agent any
  stages {
    // ... previous stages for checkout and building ...
```

```
stage('Deploy to EC2') {
        steps {
            script {
                // Configure AWS credentials (replace with placeholders or use IAM roles)
                def accessKey = 'YOUR_ACCESS_KEY_ID'
                def secretKey = 'YOUR_SECRET_ACCESS_KEY'
                // Retrieve build version from environment variable
                def buildVersion = "${BUILD_NUMBER}"

                // Use AWS CLI commands within sh steps
                sh "aws configure set aws_access_key_id ${accessKey}"
                sh "aws configure set aws_secret_access_key ${secretKey}"
                sh "aws configure set region ${AWS_DEFAULT_REGION}"

                // Login to ECR using AWS CLI
                sh "aws ecr get-login-password --region ${AWS_DEFAULT_REGION} | docker login --username AWS --password-stdin ${REPOSITORY_URI}"

                // Pull the latest image from ECR
                sh "docker pull ${REPOSITORY_URI}:${buildVersion}"

                // Access EC2 instance using SSH (replace with your details)
                sh "ssh -i <your_pem_key_file> ubuntu@<EC2_INSTANCE_PUBLIC_IP> docker stop my-web-app || true" // Stop existing container (optional)
                sh "ssh -i <your_pem_key_file> ubuntu@<EC2_INSTANCE_PUBLIC_IP> docker rm my-web-app || true"   // Remove existing container (optional)
                sh "ssh -i <your_pem_key_file> ubuntu@<EC2_INSTANCE_PUBLIC_IP> docker run -d -p 80:80 --name my-web-app ${REPOSITORY_URI}:${buildVersion}" // Run the pulled image
```

```
 // Update security group to allow access (temporary for demo)
                sh "aws ec2 authorize-security-group-ingress --group-id
<YOUR_SECURITY_GROUP_ID> --protocol tcp --port 80 --cidr 0.0.0.0/0 || true" // Allow all
traffic for demo
            }
        }
    }

    stage('Post-Build Actions') {
        steps {
            // Success message and optional reporting
            script {
                if (currentBuild.result == 'SUCCESS') {
                    echo 'Pipeline execution successful!'
                } else {
                    echo 'Pipeline execution failed!'
                }
            }
        }
    }

    post {
         always {
        emailext body: '''Job Name: ${currentBuild.fullDisplayName} Status:
${currentBuild.result} Build URL: ${env.BUILD_URL}''',
        subject: 'CI/CD Pipeline - Build Status Notification',
        to: 'qureshiabdulquader@gmail.com'
        }
    }
  }
}
```

Explanation:

- The `Checkout Code` stage retrieves code from your Git repository.
- The `Build Docker Image` stage builds the image.
- The `Deploy to EC2` stage:
    - Retrieves the ECR repository URI.
    - Configures AWS credentials temporarily for script execution (***Use IAM role.***)
    - Logs in to ECR using the AWS CLI.
    - Pulls the latest image from ECR.
    - Uses SSH to connect to the EC2 instance and performs the following actions (replace placeholders with your details):
        - Stops and removes any existing container named "my-web-app".
        - Runs the pulled image as a detached container, mapping port 80 on the container to port 80 on the EC2 instance, and naming it "my-web-app."
    - Updates the security group to allow all traffic on port 80 for demonstration purposes

7. Jenkins Job Validation and Success Message:

- We've added a new stage named "Post-Build Actions."
- Inside this stage, a script block checks the current build result (currentBuild.result).
- If the result is "SUCCESS," it displays a success message using echo. You can customize this message as needed.

8. Email Notification with Job Status:

- We've added a post section after the pipeline stages.
- Inside the always block, the emailext step sends an email notification regardless of the build outcome.
- The email body includes dynamic variables like job name, status, and build URL for easy access to details.
- And the recipient address.

9. Registering a Domain with AWS:

Steps:

1. **Access Route 53:** Go to the AWS Management Console and navigate to the Route 53 service.
2. **Register Domain:** Click on "Register domain names" and follow the wizard to register a new domain name or transfer an existing one to AWS.
3. **Create Hosted Zone:** After registration, create a Hosted Zone for your domain. This zone manages DNS records that map your domain name to resources (like the EC2 instance).
4. **Create A Record:** In the Hosted Zone, create an A record that points the domain name to your EC2 instance's public IP address. This allows users to access your application using the domain name instead of the IP address.