Phase 1: Infrastructure Setup

This phase covers the creation of the necessary cloud servers.

1. **Launch Jenkins Master & Build Slave:**
   - Launch two AWS EC2 instances (type `t2.micro` is sufficient).
   - Name them `Jenkins-Master` and `Build-Slave`.
   - Use an **Amazon Linux 2** AMI.
   - Ensure the security group allows SSH (port 22) access.
2. **Install Jenkins on Master:**
   - SSH into the `Jenkins-Master`.
   - Install Java 17: `sudo yum install java-17-amazon-corretto -y`
   - Install Jenkins and start the service as we did in the first project.
   - Complete the initial Jenkins setup wizard.
3. **Prepare Build Slave:**
   - SSH into the `Build-Slave`.
   - Install Java 17: `sudo yum install java-17-amazon-corretto -y`
   - Install Docker: `sudo yum install docker -y`
   - Start and enable Docker: `sudo systemctl start docker && sudo systemctl enable docker`
   - Add user to Docker group: `sudo usermod -aG docker ec2-user` (then log out and log back in).
   - Connect this slave to the Jenkins master using the **Nodes** menu, giving it the label `docker`.

Phase 2: Kubernetes Node & Cluster Setup

This phase prepares the environment for our application deployment.

1. **Launch Kubernetes Slave:**
   - Launch a new AWS EC2 instance.
   - **Crucially, select instance type `t2.medium`** to meet the 2 CPU core requirement for Minikube.
   - Name it `K8s-Slave`.
2. **Install Prerequisites on K8s-Slave:**
   - SSH into the `K8s-Slave`.
   - Install Java 17: `sudo yum install java-17-amazon-corretto -y`
   - Install Docker (follow the same steps as for the Build Slave).
   - Install `kubectl`:

     Bash

```
curl -LO "https://dl.k8s.io/release/$(curl -L -s
https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
chmod +x kubectl
sudo mv kubectl /usr/local/bin/
```

   - Install `minikube`:

     Bash

     ```
     curl -Lo minikube
     https://storage.googleapis.com/minikube/releases/latest/minikube-
     linux-amd64
     chmod +x minikube
     sudo mv minikube /usr/local/bin/
     ```

3. **Start the Kubernetes Cluster:**
   - Run the start command with the `--listen-address` flag. This is the key to making the cluster accessible from the EC2 instance's public IP.

     Bash

     ```
     minikube start --driver=docker --listen-address='0.0.0.0'
     ```

4. **Connect K8s-Slave to Jenkins:**
   - In the Jenkins UI, go to **Manage Jenkins -> Nodes -> New Node**.
   - Name it `K8s-Slave` and configure it as a permanent agent.
   - Use the label `kubernetes`.
   - Use the same SSH credentials and host key verification strategy as your other slaves.

Phase 3: Jenkins Pipeline Configuration

This phase configures the Jenkins jobs to build and deploy the code.

1. **Create a New GitHub Repository:**
   - Create a new repository (e.g., `jenkins-kubernetes-pipeline`).
   - Add your `Dockerfile` and the new, improved `index.html` file to it.
2. **Create the Build Job:**
   - In Jenkins, create a new Freestyle project named `K8s-Build-Job`.
   - Restrict it to run on the `docker` label.
   - Configure the **Source Code Management** section to point to your new GitHub repository.
   - In **Build Environment**, bind your Docker Hub credentials to the `DOCKER_USER` and `DOCKER_PASS` variables.
   - In the **Execute shell** build step, use this script:

Bash

```
# Build the image
docker build -t my-k8s-app .

# Tag with build number and 'latest'
docker tag my-k8s-app $DOCKER_USER/my-k8s-app:$BUILD_NUMBER
docker tag my-k8s-app $DOCKER_USER/my-k8s-app:latest

# Login and push both tags
echo $DOCKER_PASS | docker login --username $DOCKER_USER --
password-stdin
docker push $DOCKER_USER/my-k8s-app:$BUILD_NUMBER
docker push $DOCKER_USER/my-k8s-app:latest
```

3. **Create the Kubernetes Deployment Job:**
   - Create a new Freestyle project named `K8s-Deploy-Job`.
   - Restrict it to run on the `kubernetes` label.
   - In the **Execute shell** build step, use this script. This script includes the critical `rollout restart` command to defeat caching issues.

     Bash

```
# Apply the deployment manifest
kubectl apply -f - <<EOF
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-web-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-web-app
  template:
    metadata:
      labels:
        app: my-web-app
    spec:
      containers:
      - name: my-web-container
        image: your-dockerhub-username/my-k8s-app:latest
        imagePullPolicy: Always
        ports:
        - containerPort: 80
EOF

# Force a rolling update to pull the new image
echo "--- Forcing a rolling restart of the deployment ---"
kubectl rollout restart deployment my-web-app
```

Phase 4: Automation and Final Exposure

This final phase links everything together and makes the application public.

1. **Link the Jenkins Jobs:**
    - Go to the configuration for the `K8s-Build-Job`.
    - Add a **Post-build Action** of type **"Build other projects"**.
    - Enter `K8s-Deploy-Job` in the "Projects to build" field.
2. **Set Up the GitHub Webhook:**
    - In the `K8s-Build-Job` configuration, go to **Build Triggers** and check **"GitHub hook trigger for GITScm polling"**.
    - In your GitHub repository settings, go to **Webhooks** and add a new webhook.
    - The **Payload URL** must be `http://<Public_IP_of_Jenkins_Master>:8080/github-webhook/`.
3. **Expose the Application (Reliable Method):**
    - SSH into your `K8s-Slave`.
    - **Open a new, separate terminal window** and run the `kubectl port-forward` command. This must be left running.

      Bash

      ```
      kubectl port-forward deployment/my-web-app 8080:80 --
      address='0.0.0.0'
      ```

    - In your **AWS Security Group** for the `K8s-Slave`, open port `8080` to `Anywhere-IPv4`.
4. **Access Your Application:**
    - Open your browser and navigate to `http://<Public_IP_of_K8s_Slave>:8080`.