**🔬 Lab Guide 9: CI/CD with Jenkins + Docker (Staged Rollout + Rollback)**

**🎯 Learning Objectivesdy**

By the end of this lab, participants will be able to:

* Package a Python/Flask web app into a Docker image.
* Configure a Jenkins pipeline to build and deploy the app.
* Use **staging → production promotion** on the same host.
* Implement **automatic rollback** using Docker image tags (latest as last known good).
* Connect deployment practices to **SRE principles**: safe changes, fast rollback, error budget protection.

**📘 Pre-Lab Context**

Traditional deployments often overwrite what’s in production. If something breaks, rolling back is painful.

SRE best practices recommend:

* **Staging first**: test new version in a safe environment.
* **Controlled promotion**: only deploy to production after approval.
* **Rollback strategy**: always keep a stable version ready.

In Docker, rollback is easy if we:

1. Tag each build with a **unique version** (:23).
2. Tag the last stable build as **latest**.
3. If production fails, restart with :latest.

**🛠️ Prerequisites**

* Jenkins installed and running on the server ✅
* Docker installed (docker ps works as Jenkins user) ✅
* Jenkins user added to Docker group:
* sudo usermod -aG docker jenkins
* sudo systemctl restart jenkins
* GitHub repo with the app: app.py, requirements.txt, Dockerfile

**🛠️ Step 1 – Prepare Application Repository**

**Reference :** [**https://github.com/ramannkhanna2/Flask-CiCd-Jenkins-Rollout-Rollback-Docker.git**](https://github.com/ramannkhanna2/Flask-CiCd-Jenkins-Rollout-Rollback-Docker.git)

Add the following files in a GitHub repo (e.g., flask-app).

**app.py**

**requirements.txt**

**Dockerfile**

FROM python:3.12-slim

WORKDIR /app

COPY requirements.txt .

RUN pip install -r requirements.txt

COPY . .

EXPOSE 5000

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

**🛠️ Step 2 – Jenkins Job Setup**

1. Go to Jenkins → **New Item** → *Pipeline*.
2. Name it flask-pipeline.
3. Under **Pipeline → Definition**, choose **Pipeline script from SCM**.
4. Connect GitHub repo (main branch).

**🛠️ Step 3 – Jenkinsfile (Pipeline Definition)**

Create a file named Jenkinsfile in the repo root:

pipeline {

agent any

environment {

APP\_NAME = "raman-flask-app"

VERSION = "${env.BUILD\_NUMBER}"

}

stages {

stage('Build Docker Image') {

steps {

sh 'docker build -t $APP\_NAME:$VERSION .'

}

}

stage('Deploy to Staging') {

steps {

sh '''

docker rm -f ${APP\_NAME}-staging || true

docker run -d --name ${APP\_NAME}-staging -p 5001:5000 $APP\_NAME:$VERSION

echo "Staging running at http://<server-ip>:5001/"

'''

}

}

stage('Approval for Production') {

steps {

input message: "Promote this version to Production?"

}

}

stage('Deploy to Production') {

steps {

script {

try {

sh '''

# Stop old prod container but keep image

docker rm -f ${APP\_NAME}-prod || true

# Run new version

docker run -d --name ${APP\_NAME}-prod -p 5002:5000 $APP\_NAME:$VERSION

# If successful, mark this as the stable 'latest'

docker tag $APP\_NAME:$VERSION $APP\_NAME:latest

echo "Production running at http://<server-ip>:5002/"

'''

} catch (err) {

error("Production deployment failed!")

}

}

}

}

}

post {

failure {

echo "⚠️ Deployment failed! Rolling back to last stable version..."

sh '''

docker rm -f ${APP\_NAME}-prod || true

docker run -d --name ${APP\_NAME}-prod -p 5002:5000 $APP\_NAME:latest

'''

}

}

}

**🛠️ Step 4 – Test the Pipeline**

1. Commit and push code to GitHub.
2. Trigger build in Jenkins.
3. Observe pipeline:
   * Image built → runs in staging (:5001).
   * Jenkins pauses for approval.
   * On approval → runs in prod (:5002).
   * If successful → image tagged as latest for rollback safety.

**🛠️ Step 5 – Simulate Rollback**

1. Break the app ….

Edit your requirements.txt in GitHub like this:

not-a-real-package==99.99.99

Commit + push → this triggers a new Jenkins build (which will create raman-flask-app:3).

1. Jenkins builds image, deploys to staging fine.
2. When promoted to prod → container fails.
3. Rollback executes automatically:
   * Bad container removed.
   * Last good latest container restarted.
4. Production continues serving stable version.

**📊 Deliverables for Trainees**

* GitHub repo with: app.py, requirements.txt, Dockerfile, Jenkinsfile.
* Screenshot of Jenkins pipeline run (with Approval stage).
* Browser verification: staging vs production.
* Demo rollback (show prod still working after bad build).

**💡 Reflective Questions**

* Why is it important to **tag images with both version and latest**?
* How does this pipeline protect the **error budget** during bad deployments?

👉 This lab gives trainees a **hands-on end-to-end CI/CD flow with rollback** on a single Jenkins + Docker server. It’s simple, yet very close to real-world SRE practice.

**🔬 Lab Guide 10: Deploy Instances Behind an AWS Application Load Balancer**

**🎯 Learning Objectives**

By the end of this lab, participants will:

* Launch three EC2 instances running a simple web application.
* Configure an **Application Load Balancer (ALB)** in AWS.
* Connect the instances to the ALB via a Target Group.
* Verify that the ALB distributes traffic across multiple backends.

**📘 Background**

A **single server** is a single point of failure. Running **multiple backend instances** behind a Load Balancer improves:

* **Scalability** → traffic is spread across instances.
* **Fault tolerance** → if one node fails, the ALB routes traffic only to healthy ones.
* **Simplicity** → users hit one DNS endpoint, not multiple IPs.

This is a **classic HA (High Availability) pattern** in production systems.

**🛠️ Prerequisites**

* AWS account with **EC2** and **ELB** access.
* SSH key pair (.pem file).
* AWS Console access.

**🛠️ Step-by-Step Instructions**

**Step 1 – Launch Three EC2 Instances**

1. Go to **AWS Console → EC2 → Launch Instance**.
   * AMI: Amazon Linux 2.
   * Instance type: t2.micro.
   * Key pair: Select existing key.
   * Security group: Allow inbound **HTTP (80)** and **SSH (22)**.
2. Under **Advanced → User Data**, paste this script (so each instance serves a unique page):

#!/bin/bash

yum update -y

yum install -y httpd

systemctl start httpd

systemctl enable httpd

echo "<h1>Hello from Backend $(hostname)</h1>" > /var/www/html/index.html

1. Launch **three instances** using the same settings.
2. Confirm all three are **running** and note their Instance IDs.

**Step 2 – Verify Each Backend Manually**

Open the **Public IP** of each instance in a browser:

* Instance 1 → shows “Hello from Backend …”
* Instance 2 → different hostname
* Instance 3 → different hostname

👉 Confirms each server is running Apache and serving traffic.

**Step 3 – Create a Target Group**

1. In AWS Console → **EC2 → Target Groups → Create Target Group**.
   * Target type: **Instances**.
   * Protocol: **HTTP**.
   * Port: **80**.
   * Health Check Path: /.
2. Register your **3 EC2 instances** in this target group.

**Step 4 – Create an Application Load Balancer (ALB)**

1. Go to **EC2 → Load Balancers → Create Load Balancer → Application Load Balancer**.
   * Name: app-lb-lab.
   * Scheme: Internet-facing.
   * Listener: HTTP (80).
   * Availability Zones: Select at least 2 AZs (recommended).
2. Under **Listeners and Routing**, forward traffic to the **Target Group** you created.
3. Finish creation.

**Step 5 – Verify Load Balancer is Working**

1. Once ALB is active, AWS provides a **DNS name** like:
2. app-lb-lab-123456.ap-south-1.elb.amazonaws.com
3. Open this DNS in your browser.
   * Refresh several times → you should see responses alternating between:
     + “Hello from Backend 1”
     + “Hello from Backend 2”
     + “Hello from Backend 3”

👉 Confirms ALB is distributing requests across all 3 backends.

**Step 6 – Simulate a Node Failure (Optional Extension)**

1. SSH into one backend:
2. sudo systemctl stop httpd

(Simulates an app crash).

1. Wait 1–2 minutes → ALB health check marks the instance **Unhealthy**.
2. Refresh ALB DNS in browser → only the **2 healthy nodes** respond.
3. Restart service to recover:
4. sudo systemctl start httpd

**📊 Analysis (SLIs/SLOs)**

* **Availability SLI:** Ratio of successful responses from ALB.
* **SLO Example:** 99.9% availability per month.
* **Error Budget Impact:** A single-node failure does **not** impact SLO thanks to ALB routing.