**Topics**

1. **Introduction to Continuous Delivery (CD)**
   * **Continuous Delivery (CD)** is the practice of automating the delivery of software to production or staging environments after every successful build and test.
   * While **Continuous Integration (CI)** focuses on integrating changes into a shared repository, **Continuous Delivery** ensures that those changes can be automatically deployed to production at any time.
2. **CD Pipeline Overview**
   * A **CD pipeline** is a series of automated steps that deploy your application to the intended environment.
   * Key components of a CD pipeline:
     + **Build**: Compiling code and generating build artifacts.
     + **Test**: Running unit, integration, and UI tests.
     + **Deploy**: Deploying the application to a staging or production environment.
     + **Monitor**: Verifying the health of the deployment.
3. **Automating Deployments Using Python**
   * Use Python scripts to automate deployments and handle tasks like uploading files, restarting servers, and interacting with APIs to trigger deployments.
   * **Deployment tools**: Use Python to interface with deployment tools like **Ansible**, **Terraform**, or simply run shell commands to deploy applications.
4. **CD with Python and GitHub Actions**
   * Leverage **GitHub Actions** or other CI tools to implement Continuous Delivery pipelines with Python.
   * Automating the entire process from code commit to deployment can be achieved using a combination of GitHub Actions, Python scripts, and cloud services.

**Practical Exercises**

**1. Setting Up Continuous Delivery with GitHub Actions**

* **Create a deploy.yml GitHub Actions Workflow**: A GitHub Actions workflow for CD typically runs after tests pass and the code is successfully built. You can use the same repository that you set up for CI.

Example deploy.yml for deploying an application:

yaml

Copy code

name: Continuous Delivery Pipeline

on:

push:

branches:

- main # Trigger deployment on push to the 'main' branch

pull\_request:

branches:

- main

jobs:

deploy:

runs-on: ubuntu-latest

steps:

- name: Checkout code

uses: actions/checkout@v2

- name: Set up Python

uses: actions/setup-python@v2

with:

python-version: '3.8'

- name: Install dependencies

run: |

pip install -r requirements.txt

- name: Deploy to production

run: |

python deploy.py

**Explanation**: The deploy.yml file triggers whenever changes are pushed to the main branch, installs dependencies, and then runs a deploy.py script to deploy the application.

**2. Automating Deployment with Python Script (deploy.py)**

* **Deployment Script Example**: Let’s create a Python script (deploy.py) that automates deployment. This could involve copying files, restarting services, or interacting with cloud platforms.

Example deploy.py for deploying an application to a remote server using SSH:

* 1. **Install the paramiko library** to SSH into remote servers:

bash

Copy code

pip install paramiko

* 1. **Create a deploy.py script**:

python

Copy code

import paramiko

def deploy():

# SSH client to connect to the remote server

ssh = paramiko.SSHClient()

ssh.set\_missing\_host\_key\_policy(paramiko.AutoAddPolicy())

# Remote server details

hostname = "your-server-ip"

username = "your-username"

password = "your-password" # Or use SSH keys

# Connect to the remote server

ssh.connect(hostname, username=username, password=password)

# Run deployment commands (e.g., pulling latest changes, restarting services)

commands = [

"cd /path/to/your/app",

"git pull origin main",

"sudo systemctl restart your-service"

]

for command in commands:

stdin, stdout, stderr = ssh.exec\_command(command)

print(f"Executing: {command}")

print(stdout.read().decode())

print(stderr.read().decode())

# Close the SSH connection

ssh.close()

if \_\_name\_\_ == '\_\_main\_\_':

deploy()

**Explanation**:

* 1. The script connects to a remote server via SSH.
  2. It pulls the latest changes from the GitHub repository and restarts a service to apply the new changes.
  3. You can modify the commands based on your specific deployment environment (e.g., restarting Docker containers, deploying to cloud platforms).

**3. Automating Rollbacks Using Python**

* **Rollback Strategy**: It’s essential to automate rollback procedures in case a deployment fails. You can create a Python script to automatically rollback to a previous version of the application.

Example rollback functionality:

python

Copy code

import paramiko

def rollback():

ssh = paramiko.SSHClient()

ssh.set\_missing\_host\_key\_policy(paramiko.AutoAddPolicy())

# Remote server details

hostname = "your-server-ip"

username = "your-username"

password = "your-password" # Or use SSH keys

ssh.connect(hostname, username=username, password=password)

# Commands to rollback deployment

rollback\_commands = [

"cd /path/to/your/app",

"git checkout HEAD~1", # Roll back to the previous commit

"sudo systemctl restart your-service"

]

for command in rollback\_commands:

stdin, stdout, stderr = ssh.exec\_command(command)

print(f"Executing: {command}")

print(stdout.read().decode())

print(stderr.read().decode())

ssh.close()

if \_\_name\_\_ == '\_\_main\_\_':

rollback()

**Explanation**: The script connects to a remote server, rolls back to the previous commit using git checkout HEAD~1, and restarts the service to apply the changes. This is useful in case the latest deployment introduces issues.

**4. Deployment with Cloud Platforms (AWS, GCP, Azure)**

* **Deploying to AWS EC2 Using Python**:
  + First, install boto3, the AWS SDK for Python:

bash

Copy code

pip install boto3

* + Example Python script to deploy to an EC2 instance:

python

Copy code

import boto3

def deploy\_to\_aws():

ec2 = boto3.client('ec2')

# Start the instance (replace 'i-xxxxxxxxxxxxxxxxx' with your instance ID)

response = ec2.start\_instances(InstanceIds=['i-xxxxxxxxxxxxxxxxx'])

print(response)

if \_\_name\_\_ == '\_\_main\_\_':

deploy\_to\_aws()

**Explanation**: This script uses boto3 to interact with AWS EC2. It starts an EC2 instance (or you can modify it to deploy your app to the instance).

**5. Continuous Deployment with GitHub Actions and Python**

* **GitHub Actions + Python for Full Continuous Delivery**: In combination with the earlier deploy.yml GitHub Actions workflow, you now have an automated deployment pipeline:
  + Push code to main → Trigger GitHub Actions.
  + GitHub Actions installs dependencies, runs tests, and executes deploy.py for deployment.
  + Optionally, if the deployment fails, trigger a rollback using the rollback script.

The full process is automated, ensuring that your application is continuously delivered to production without manual intervention.

**6. Challenge: Build and Automate a Full CD Pipeline**

* **Task**:
  + Create a full Continuous Delivery pipeline using Python.
  + Use GitHub Actions to automate the deployment of an application to a remote server or cloud platform.
  + Implement rollbacks for failed deployments.
  + Use environment variables for sensitive data like API keys or SSH credentials.

**Summary**

* **Day 10** focused on **Continuous Delivery (CD)** and automating deployment processes using **Python** and **GitHub Actions**.
* You learned how to set up a **CD pipeline** with GitHub Actions, deploy applications using Python scripts, automate rollbacks, and even interact with cloud services like AWS.
* Automating your deployment pipeline ensures quick and reliable delivery of software updates to production.