CI/CD Fundamentals

for Modern Banking Applications

Building Reliable and Secure Personal Banking Services

ci-demo-python.zip

Agenda

🎯 **The 'Why': Understanding Basic DevOps Terms**

🔄 **The Core Concepts:**

- Continuous Integration (CI)
- Continuous Delivery (CD)
- Continuous Deployment (CD)

💻 **Hands-on Exercises:**

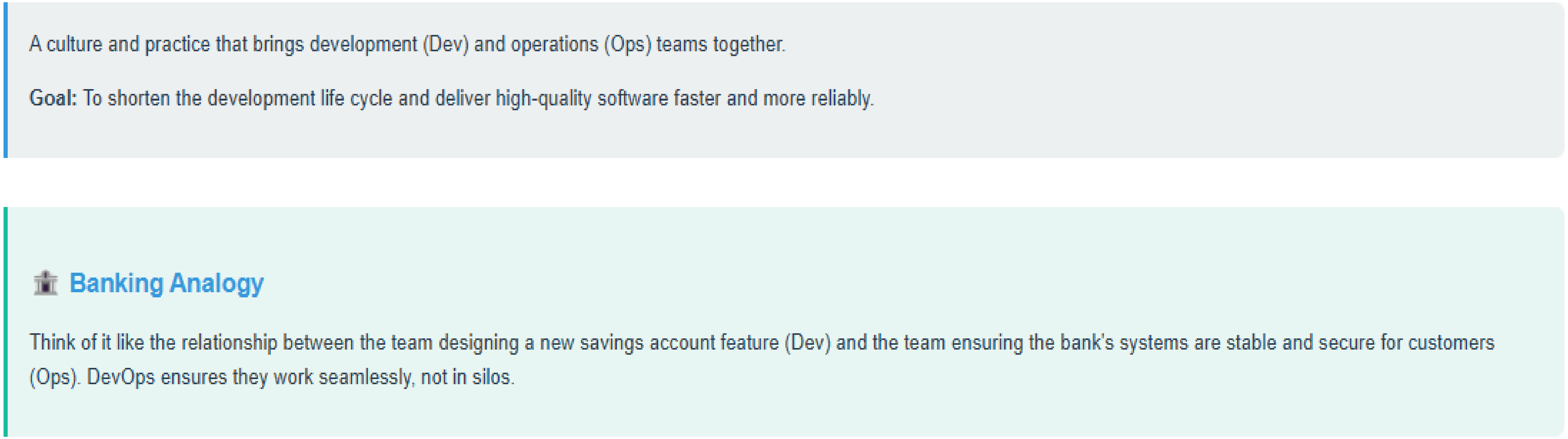
- Creating a Python CI Pipeline

🚀 **Best Practices & Pro-Tips for FinTech**

✅ **Conclusion & Key Takeaways**

Understanding DevOps Terms

What is DevOps?



A culture and practice that brings development (Dev) and operations (Ops) teams together.

Goal: To shorten the development life cycle and deliver high-quality software faster and more reliably.

🏦 Banking Analogy

Think of it like the relationship between the team designing a new savings account feature (Dev) and the team ensuring the bank's systems are stable and secure for customers (Ops). DevOps ensures they work seamlessly, not in silos.

Continuous Delivery (CD)

What is it?

An extension of Continuous Integration. Every code change that passes the automated tests is automatically released to a testing or "staging" environment.

Why is it important in banking?

- **Always Release-Ready:** You can deploy a new feature or a critical security patch at any time with the click of a button.
- **Lower Risk:** Deploying smaller changes is less risky than deploying one massive update.
- **Manual Approval Gate:** The final push to customers (production) requires a manual approval. This is crucial for banking to allow for final compliance and business checks.

CD in Action

The "Mobile Check Deposit" feature passes all tests. It is then automatically deployed to a staging server where the QA team and product managers can test it in a production-like environment.

Continuous Deployment (CD)

What is it?

The next step after Continuous Delivery. Every change that passes all stages of your production pipeline is released directly to your customers. There's no human intervention.

Is this suitable for banking?

- **High-Risk:** Due to strict regulatory and security requirements, true Continuous Deployment is rare in core banking systems.
- **Possible for Some Services:** It might be used for less critical, non-transactional parts of the application, like updating marketing content or a help page.

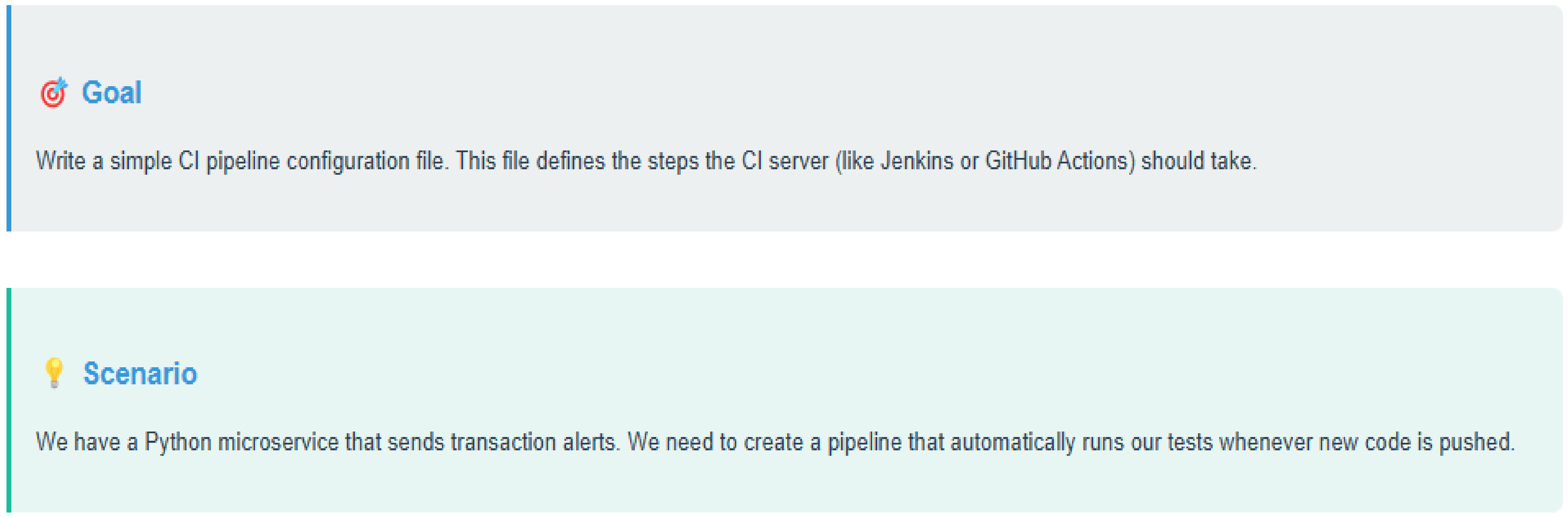
The Key Difference:

Continuous Delivery
Manual approval before production

VS

Continuous Deployment
No manual approval needed

Hands-on Exercise #1: Python CI Pipeline



🎯 Goal

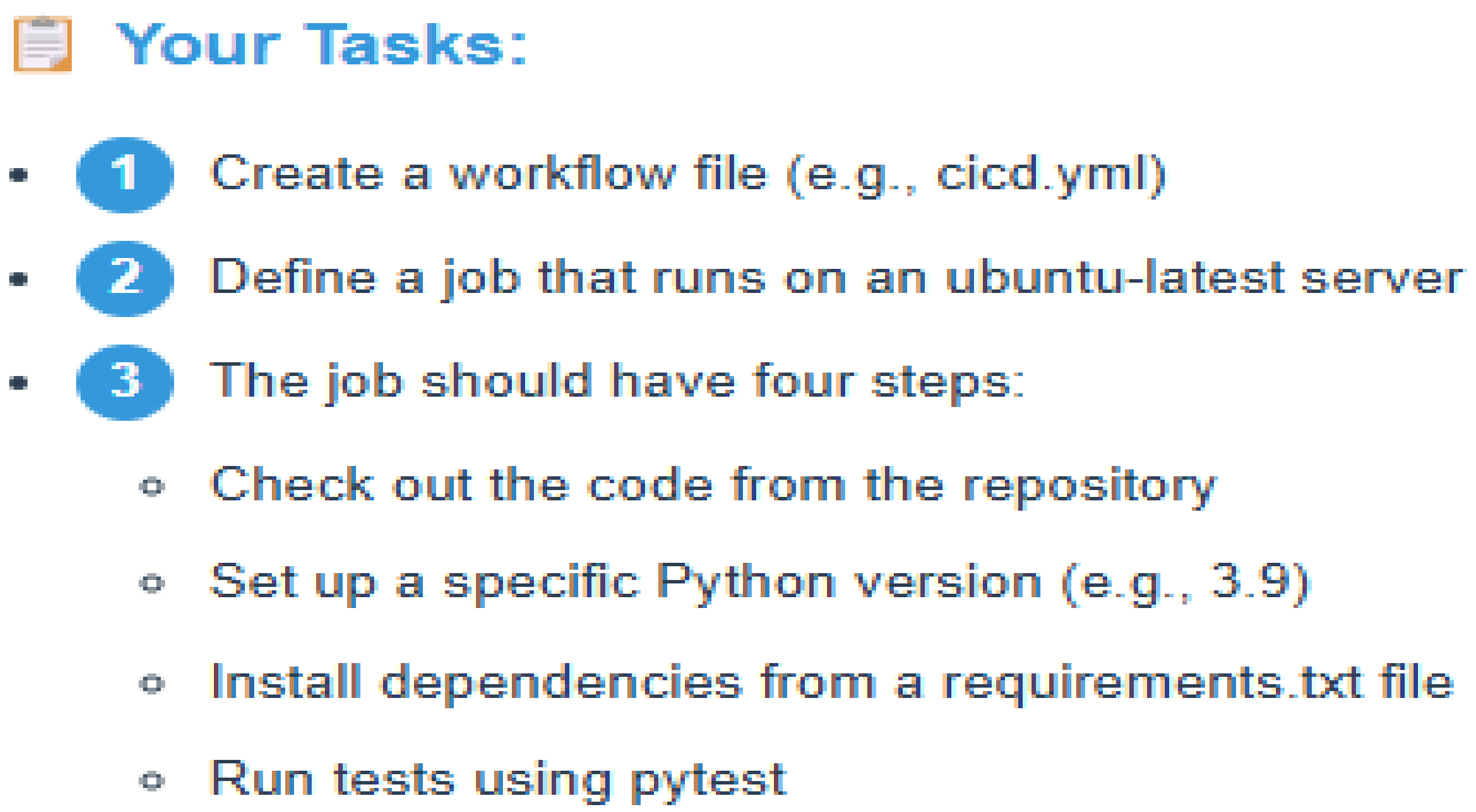
Write a simple CI pipeline configuration file. This file defines the steps the CI server (like Jenkins or GitHub Actions) should take.

💡 Scenario

We have a Python microservice that sends transaction alerts. We need to create a pipeline that automatically runs our tests whenever new code is pushed.

Exercise #2 - The Task (Python)

We will use the GitHub Actions format.



📋 Your Tasks:

- 1 Create a workflow file (e.g., cicd.yml)
- 2 Define a job that runs on an ubuntu-latest server
- 3 The job should have four steps:
  - Check out the code from the repository
  - Set up a specific Python version (e.g., 3.9)
  - Install dependencies from a requirements.txt file
  - Run tests using pytest

Exercise #2 - Solution (Python)

This is what the cicd.yml file would look like:

```yaml
# .github/workflows/cicd.yml
name: Python Application CI

on: [push] # Run this workflow on every push to the repository

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
    - name: Check out code
      uses: actions/checkout@v3

    - name: Set up Python 3.9
      uses: actions/setup-python@v4
      with:
        python-version: '3.9'

    - name: Install Dependencies
      run: |
        python -m pip install --upgrade pip
        pip install -r requirements.txt

    - name: Run Tests with Pytest
      run: |
        pytest
```

💭 **Discussion: What would happen if the pytest step fails?**

on: [push]

The trigger. Tells the robot chef, "Start cooking on every push event."

runs-on: ubuntu-latest

Uses a fresh, virtual computer running the latest version of Ubuntu for this job.

Step 1: Check out code

uses: actions/checkout@v3

Robot Chef says: "First, I need to get all the ingredients (your source code) from the pantry (the repository) and place them on my kitchen counter (the Ubuntu machine)."

Step 2: Set up Python

uses: actions/setup-python@v4

Robot Chef says: "Now, I need to get the right tool for the job. This recipe requires a Python 3.11 mixer."

Step 3: Install Dependencies

Robot Chef says: "The recipe requires some special spices listed in the requirements.txt file. I'll install them now."

Uses pip (Python's package installer) to install all required libraries.

Step 4: Run Tests with Pytest

Robot Chef says: "Time for a taste test! I will run pytest to make sure the dish tastes perfect (the code works as expected)."

If any test fails, the entire workflow fails, and you get a notification! 🚨

Knowledge Check: Q&A (Part 1)

Q1: What is the main purpose of this workflow file?

To automate the process of building and testing our Python application every time new code is pushed to the repository. This is called Continuous Integration (CI).

Q2: What event triggers this workflow to run?

The push event.

Q3: In our robot chef analogy, what does runs-on: ubuntu-latest represent?

It represents the kitchen or environment where the robot chef works—a virtual machine running the latest version of Ubuntu.

Knowledge Check: Q&A (Part 2)

Q4: Why is the actions/checkout@v3 step necessary?
It's necessary to download the source code from the repository onto the virtual machine so the workflow can access and test it.

Q5: If you wanted to use Python version 3.12 instead of 3.11, which line would you change?
You would change the python-version: '3.11' line to python-version: '3.12'.

Q6: What command is used to run the tests in this workflow?
pytest

Hands-On Classroom Exercise: Add a Linter!

Goal: Add a new step to the workflow that checks the code for style issues using a tool called **Flake8**. A linter is like a grammar checker for your code.

Step 1: Prepare Your Repository

1.Create a new repository on GitHub.
2.Create a simple Python file named `app.py` with some code (it can even have a deliberate style error).

```
# app.py
def add(a, b):
  return a+b # Deliberate style error: no space around ‘+’
```

3. Create a requirements.txt file and add the libraries you'll need.

```
# requirements.txt
pytest
Flake8
```

4. Create a test file test_app.py.

```
# test_app.py
from app import add

def test_add():
    assert add(2, 3) == 5
```

5. Commit and push these files to your repository.

Step 2: Add the Workflow File

1. In your repository, go to the "Actions" tab. GitHub might suggest a Python workflow. Click "Configure" or create one yourself.

2. Create a file at this path: `.github/workflows/ci.yml`.

3. Copy the original workflow content from the presentation into this file.

Step 3: Modify the Workflow - Add the Linter Step!

```
# ... (previous steps) ...

 - name: Install Dependencies
   run: |
     python -m pip install --upgrade pip
     pip install -r requirements.txt

 # --- ADD THIS NEW STEP ---
 - name: Lint with Flake8
   run: |
     # stop the build if there are Python syntax errors or undefined names
     flake8 . --count --select=E9,F63,F7,F82 --show-source --statistics
     # exit-zero treats all errors as warnings. The GitHub editor is 127 chars wide
     flake8 . --count --exit-zero --max-complexity=10 --max-line-length=127 --statistics

 - name: Run Tests with Pytest
   run: |
     pytest
```

Step 4: See it in Action!

1. Commit and push your updated `ci.yml` file.
2. Go to the "Actions" tab in your GitHub repository.
3. Click on your new workflow run. You should see the new "Lint with Flake8" step. Watch it run! If your code has style issues, `flake8` will report them in the log.

💭 Discussion: What would happen if the pytest step fails?

Best Practices & Pro-Tips for Banking

🔒 Security First (DevSecOps)

Integrate static code analysis (SAST) and dependency scanning tools directly into the CI pipeline.

Pro-Tip: Use tools like SonarQube or Snyk to automatically flag security vulnerabilities before they reach production.

⚡ Keep Builds Fast

A core principle of CI is fast feedback. If a build takes an hour, developers will be less productive.

Pro-Tip: Parallelize your tests and use caching for dependencies to speed up pipeline execution.

🏗 Infrastructure as Code (IaC)

Define your testing and staging environments in code (e.g., using Terraform or CloudFormation). This ensures consistency and auditability.

Pro-Tip: The CI/CD pipeline should create and tear down these environments automatically for each feature branch.

📋 Comprehensive Audit Trails

Every action, from code commit to deployment, must be logged. This is non-negotiable for regulatory compliance (e.g., SOX).

Pro-Tip: Your CI/CD tool should integrate with logging systems and require authentication for all manual approval gates.

Conclusion & Key Takeaways

🚀 CI/CD is a journey, not a destination

It's about continuous improvement.

🎯 Key Takeaways:

- **Automation is Key:** Automating builds and tests reduces human error, which is critical when dealing with financial data.
- **Security is Not an Afterthought:** It must be integrated into every step of the pipeline.
- **For Banking, Continuous Delivery is the Sweet Spot:** It provides the benefits of automation while keeping a crucial human approval step for safety and compliance.

Code → Build → Test → Deploy