**Lab Manual**

**Building Your First Python CI Pipeline with GitHub Actions**

**Objective**

By the end of this lab, you should be able to:

* Set up **GitHub Actions** to automatically test your Python programs.
* Understand the purpose of each step in a workflow file.
* Gain practical experience with a simple **Continuous Integration (CI)** pipeline using **GitHub Desktop** and **Visual Studio Code**.

**Goal**

Your goal is to make sure that every time you push code to GitHub, your tests run automatically.  
This helps you catch errors early and keep your project stable and working.

**What is Continuous Integration (CI)?**

Continuous Integration is like having a built-in code reviewer that never sleeps.  
Each time you add or modify code:

* It installs all required packages,
* Runs your test cases, and
* Reports whether anything broke.

No more “It works on my computer!” problems.  
GitHub runs your code on its own cloud machines to ensure it works everywhere.

**Before You Start**

Make sure you have:

* A **GitHub account**
* **Python 3** installed on your computer
* **GitHub Desktop**
* **Visual Studio Code (VS Code)** with the Python extension
* Internet access

**Step 1 – Create and Clone a Repository**

1. Open **GitHub Desktop** → click **File → New Repository**.
2. Name it something like python-ci-demo.
3. Choose a local folder and click **Create Repository**.
4. Click **Publish Repository** to upload it to GitHub (choose Public or Private).
5. The repository is now connected to GitHub.

Any code you add here in VS Code can be pushed online directly through GitHub Desktop.

**Step 2 – Add Python Code**

Open the project in **VS Code**.

Create a file named **app.py** and add:

def add(a, b):

return a + b

Create a folder called **tests** and inside it a file named **test\_app.py**.  
Add:

from app import add

def test\_add():

assert add(2, 3) == 5

Create a **requirements.txt** file and add:

pytest

Add an empty file named **\_\_init\_\_.py** in both your main folder and the tests/ folder.  
This tells Python to treat these folders as packages so imports work correctly.

Your folder structure should now look like this:

python-ci-demo/

│

├── app.py

├── requirements.txt

├── \_\_init\_\_.py

└── tests/

├── \_\_init\_\_.py

└── test\_app.py

**Step 3 – Create the GitHub Actions Workflow**

GitHub Actions looks for configuration files in a special directory called .github/workflows.

1. In **VS Code**, create the folders:
2. .github/workflows/
3. Inside that folder, create a new file called **ci.yml**.
4. Paste the following content:

name: Python CI Pipeline

on:

push:

branches: [ main ]

pull\_request:

branches: [ main ]

jobs:

build:

runs-on: ubuntu-latest

steps:

- name: Checkout code

uses: actions/checkout@v3

- name: Set up Python

uses: actions/setup-python@v4

with:

python-version: '3.x'

- name: Install dependencies

run: |

python -m pip install --upgrade pip

pip install -r requirements.txt

- name: Run tests

run: |

export PYTHONPATH=$PWD

pytest --maxfail=5 --disable-warnings

**Purpose of This Workflow**

To automate testing of your Python project so that:

* Each new commit or pull request is checked automatically, and
* You’re alerted immediately if something breaks.

No more “it works on my machine” — GitHub runs your code on a fresh cloud server each time.

**Line-by-Line Explanation**

**1️⃣ Workflow Name**

**name: Python CI Pipeline**

* This gives your workflow a name that appears in the GitHub Actions tab (e.g., “Python CI Pipeline”).
* It helps you identify what this workflow does.

**2️⃣ When Should the Workflow Run**

**on:**

**push:**

**branches: [ main ]**

**pull\_request:**

**branches: [ main ]**

* This section defines triggers — the events that start the workflow.
* It runs automatically whenever:
  + Code is pushed to the main branch, or
  + A pull request is made into the main branch.
* That means any update to your main codebase gets tested immediately.

**3️⃣ Jobs Section**

jobs:

build:

runs-on: ubuntu-latest

* A workflow can have multiple “jobs.”
* Here, there’s one job called build.
* runs-on: ubuntu-latest tells GitHub to use a virtual machine (runner) with Ubuntu Linux installed to execute your job.
* Each job runs in a clean, isolated environment every time — so it’s consistent and reproducible.

**4️⃣ Step 1: Checkout Code**

- name: Checkout code

uses: actions/checkout@v3

* Downloads your repository’s code into the runner.
* Without this, the VM wouldn’t have access to your files (like app.py or requirements.txt).
* This is almost always the first step in a workflow.

**5️⃣ Step 2: Set Up Python**

- name: Set up Python

uses: actions/setup-python@v4

with:

python-version: '3.x'

* Installs Python 3 (any 3.x version) on the runner.
* This ensures your tests run in the correct environment — just like on your local machine.
* You can specify a version (e.g., '3.10') if your project requires it.

**6️⃣ Step 3: Install Dependencies**

- name: Install dependencies

run: |

python -m pip install --upgrade pip

pip install -r requirements.txt

* Runs two shell commands:
  1. Updates pip to the latest version.
  2. Installs all libraries listed in requirements.txt (e.g., pytest, flask, etc.).
* This ensures the test environment has everything your app needs to run.

**7️⃣ Step 4: Run Tests**

- name: Run tests

run: |

export PYTHONPATH=$PWD

pytest --maxfail=5 --disable-warnings

* Runs all tests in your project using pytest.
* export PYTHONPATH=$PWD ensures Python can locate your app modules.
* --maxfail=5 stops after 5 test failures (to save time).
* --disable-warnings hides unnecessary warning messages for a cleaner log.

If all tests pass ✅ — GitHub marks the workflow as successful.  
If any test fails ❌ — it stops and shows error logs, helping you debug.

**Summary Table**

| **Section** | **Purpose** | **Key Concept** |
| --- | --- | --- |
| name: | Labels the workflow | Easy identification |
| on: | Defines triggers | Runs on push or PR |
| jobs: | Groups tasks | Defines environments & steps |
| checkout | Downloads repo | Makes files available |
| setup-python | Installs Python | Prepares runtime |
| install dependencies | Installs packages | Matches dev setup |
| run tests | Executes pytest | Validates code quality |

**End Result**

Whenever you push or submit a pull request:

1. GitHub spins up a fresh Ubuntu server.
2. Installs Python + dependencies.
3. Runs all your tests.
4. Reports the result directly in your repo’s Actions tab or pull request view.

You now have a fully automated testing pipeline — a key foundation for professional CI/CD systems.

**Step 4 – Commit and Push Using GitHub Desktop**

1. Open **GitHub Desktop**.
2. You will see your modified files listed.
3. Type a short commit message such as:
4. Added CI workflow using GitHub Actions
5. Click **Commit to main** → **Push origin**.

**Step 5 – Check the Workflow on GitHub**

1. Go to your repository on **GitHub.com**.
2. Click the **Actions** tab.
3. A new workflow named **Python CI Pipeline** will appear.
4. Wait for all steps to turn green ✅.
   * If any step fails, click it to view the error details.

**Step 6 – Try Extensions**

Once your base pipeline works, you can add a few more useful steps.

**1. Code Style Check (Linting)**

- name: Lint with flake8

run: |

pip install flake8

flake8 .

**Purpose:**

This step checks whether your Python code follows standard style conventions defined by PEP 8.

flake8 scans every .py file and reports problems such as extra spaces, missing blank lines, or poorly formatted code.

**Example:**

Suppose you have this file:

def add(a, b):

return a + b

flake8 might report:

./app.py:3:1: W293 blank line contains whitespace

This means your “blank” line actually contains invisible spaces or tabs.

Simply delete those spaces so the blank line is truly empty:

After fixing, flake8 will run cleanly with no output.

**Another common example:**

./tests/test\_app.py:3:1: E302 expected 2 blank lines, found 1

This means your function is too close to the import statement.

Fix it by adding one more blank line before the function:

from app import add

def test\_add():

assert add(2, 3) == 5

flake8 helps maintain a consistent, professional code format across the entire project — just like real-world development teams do.

**2. Code Coverage**

- name: Test coverage

run: |

pip install pytest-cov

pytest --cov=app tests/

**Purpose:**  
This step checks how much of your code is actually being tested.  
It shows what percentage of your functions or lines were executed when the tests ran.

**Example Output:**

----------- coverage: platform linux, python 3.11 -----------

Name Stmts Miss Cover

---------------------------------

app.py 2 0 100%

---------------------------------

TOTAL 2 0 100%

A **100 % coverage** means all lines in your program ran during the tests.  
If you see less than 100 %, it means there’s code that never gets tested — a great opportunity to add more test cases.

**Step 7 – Verify Your Results**

When everything runs successfully, you should see something like this in your Action logs:

================== test session starts ==================

collected 1 item

tests/test\_app.py ....... [100%]

=================== 1 passed in 0.05s ====================

That means your setup works perfectly!

**Step 8 – Wrap Up**

From now on, whenever you push new code:

1. GitHub starts a new cloud environment.
2. Installs your dependencies.
3. Runs your tests automatically.
4. Displays the status (✅ or ❌) in the repository.

You have successfully created a **Continuous Integration pipeline**—the same concept used in modern software companies.

**Lab Deliverables**

Upload the following to Canvas:

1. A screenshot showing your successful CI workflow on GitHub.
2. A screenshot of your pytest output (showing “1 passed”).
3. The link to your GitHub repository.

**Key Takeaways**

* You set up automated testing using GitHub Actions.
* You learned how to configure and interpret each workflow step.
* You used **GitHub Desktop** and **VS Code** to manage your code and commits.
* You now understand the basic structure of professional CI/CD workflows.

**Part 2: Continuous Deployment (CD) with GitHub Actions**

**Objective**

To extend your CI pipeline into a **CI/CD workflow** — where:

* CI (Continuous Integration) = test and validate your code automatically.
* CD (Continuous Deployment) = deploy your app automatically after tests pass.

After this lab, your students will:  
✅ Understand deployment pipelines  
✅ Push code to GitHub and watch it go live automatically  
✅ Learn how to safely use GitHub secrets for credentials

**What is Continuous Deployment (CD)?**

**Continuous Deployment** means your app moves from *“tested and verified”* → *“live and running”* automatically.

For example:

You push code → GitHub Actions tests it → if all tests pass → the app is deployed to a server or container registry.

This removes manual deployment steps and ensures fast, reliable releases.

**Pre-requisites**

Before starting:

* Complete **Part 1 (CI Setup)**.
* Have a working **Python app** (Flask/FastAPI/etc.) with app.py.
* Have Docker installed **(optional, for container-based deployment)**.
* Have an **AWS account** or **Docker Hub account** (we’ll do both options).

**Deploy to Docker Hub**

**Step 1: Create a Dockerfile**

In your project root, create a Dockerfile:

# Use Python base image

FROM python:3.10-slim

# Set working directory

WORKDIR /app

# Copy project files

COPY . /app

# Install dependencies

RUN pip install --no-cache-dir -r requirements.txt

# Expose default Flask port

EXPOSE 5000

# Run the app

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

**What is a Dockerfile?**

A **Dockerfile** is a blueprint for building a Docker image — a lightweight, standalone package that contains everything your application needs to run:

* Your code
* Dependencies (libraries, packages)
* System configurations

Once built, this image can run **anywhere**, ensuring “it works on my machine” problems disappear.

**Line-by-Line Explanation**

**1️⃣ Base Image**

FROM python:3.10-slim

* This tells Docker to **start from an existing image** that already has **Python 3.10** pre-installed.
* The -slim version is a lightweight variant — smaller and faster to build — containing only essential components.
* It saves you time because you don’t have to install Python manually inside the container.

**2️⃣ Set the Working Directory**

WORKDIR /app

* This command creates (if needed) and switches to the /app directory inside the container.
* Every command that follows (like COPY, RUN, or CMD) will execute from this directory.

**3️⃣ Copy Your Project Files**

COPY . /app

* Copies all files and folders from your **local project directory** (where the Dockerfile lives) into /app inside the container.
* This includes your app.py, requirements.txt, and any other necessary project files.

**4️⃣ Install Dependencies**

RUN pip install --no-cache-dir -r requirements.txt

* Runs a command **inside the container** to install dependencies listed in requirements.txt.
* --no-cache-dir avoids storing temporary installation files, reducing image size.

**Example:**  
If your requirements.txt contains:

Flask

pytest

this line will install both inside the container.

**5️⃣ Expose the Application Port**

EXPOSE 5000

* Informs Docker that the app inside the container will listen on **port 5000**.
* This is the **default port for Flask apps**.
* Note: This doesn’t *open* the port by itself; it’s a declaration used when running the container (e.g., -p 5000:5000).

**6️⃣ Run the Application**

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

* Specifies the command to **start your app** when the container runs.
* This executes:
* python app.py
* CMD runs **only once** — when the container starts — not during image build.

**Step 2: Create Docker Hub Account and Access Token**

1. Go to [Docker Hub](https://hub.docker.com/).
2. Create an account and a repository (e.g., python-ci-demo).
3. Go to **Account Settings → Security → New Access Token**.
   * Copy the token (you’ll use it as a secret in GitHub).-

A screenshot of a computer

AI-generated content may be incorrect.

**Step 3: Add GitHub Secrets**

In your GitHub repo:

1. Go to **Settings → Secrets and variables → Actions**.
2. Click **New repository secret** for each:
   * DOCKERHUB\_USERNAME → your Docker Hub username
   * DOCKERHUB\_TOKEN → the token you copied

A screenshot of a computer

AI-generated content may be incorrect. **Step 4: Update Your Workflow File**

Modify .github/workflows/ci.yml → rename it to ci-cd.yml or add a second job for deployment:

name: Python CI/CD Pipeline

on:

push:

branches: [ main ]

pull\_request:

branches: [ main ]

jobs:

build:

runs-on: ubuntu-latest

steps:

- name: Checkout code

uses: actions/checkout@v3

- name: Set up Python

uses: actions/setup-python@v4

with:

python-version: '3.x'

- name: Install dependencies

run: |

python -m pip install --upgrade pip

pip install -r requirements.txt

- name: Run tests

run: |

pytest --maxfail=5 --disable-warnings

deploy:

needs: build

runs-on: ubuntu-latest

if: success() # Only run if tests pass

steps:

- name: Checkout code

uses: actions/checkout@v3

- name: Log in to Docker Hub

uses: docker/login-action@v3

with:

username: ${{ secrets.DOCKERHUB\_USERNAME }}

password: ${{ secrets.DOCKERHUB\_TOKEN }}

- name: Build and push Docker image

uses: docker/build-push-action@v5

with:

context: .

push: true

tags: ${{ secrets.DOCKERHUB\_USERNAME }}/python-ci-demo:latest

**Step-by-Step Explanation**

Excellent — here’s a clear, student-friendly **explanation and breakdown** of that **GitHub Actions CI/CD workflow file (ci-cd.yml)**, line by line.

**Purpose of the Workflow**

This YAML file defines a **Continuous Integration (CI)** and **Continuous Deployment (CD)** pipeline for a Python project.

* **CI (Continuous Integration)** → Tests the code automatically every time new code is pushed or a pull request is created.
* **CD (Continuous Deployment)** → If all tests pass, it automatically builds and publishes a Docker image to Docker Hub.

**Step-by-Step Explanation**

**1️⃣ Workflow Name**

name: Python CI/CD Pipeline

This gives your workflow a descriptive name — it will appear as *“Python CI/CD Pipeline”* under the **Actions** tab on GitHub.

**2️⃣ Trigger Events**

on:

push:

branches: [ main ]

pull\_request:

branches: [ main ]

This section defines **when** the workflow should run:

* Every time code is **pushed** to the main branch.
* Every time someone opens or updates a **pull request** targeting the main branch.

This ensures the workflow runs automatically for both direct commits and collaborative merges.

**3️⃣ Jobs Section**

All automation tasks are grouped under **jobs**.  
Each job runs independently in a clean virtual machine.

There are **two jobs here:**

1. build – for testing (CI)
2. deploy – for deployment (CD)

**Job 1: Build (Continuous Integration)**

build:

runs-on: ubuntu-latest

This line tells GitHub to use a **fresh Ubuntu virtual machine** as the environment.

**Steps inside the build job**

**a. Checkout the repository**

- name: Checkout code

uses: actions/checkout@v3

This pulls your project files from GitHub into the runner’s environment so the next steps can access them.

**b. Set up Python**

- name: Set up Python

uses: actions/setup-python@v4

with:

python-version: '3.x'

This installs Python (version 3.x) on the runner.  
It ensures the right version is available to run your code and tests.

**c. Install dependencies**

- name: Install dependencies

run: |

python -m pip install --upgrade pip

pip install -r requirements.txt

This installs your project’s dependencies listed in requirements.txt, such as pytest or flake8.

**d. Run tests**

- name: Run tests

run: |

pytest --maxfail=5 --disable-warnings

This executes your test suite using pytest.  
If any test fails, the pipeline will stop, marking the build as **failed**.  
Only if all tests pass will the next job (“deploy”) run.

**Job 2: Deploy (Continuous Deployment)**

deploy:

needs: build

runs-on: ubuntu-latest

if: success() # Only run if tests pass

* needs: build → means **this job depends on the “build” job**.  
  It will start **only if the build job succeeds**.
* if: success() → is an extra safety check ensuring deployment runs only after successful testing.

**Steps inside the deploy job**

**a. Checkout code again**

- name: Checkout code

uses: actions/checkout@v3

The runner is a new machine, so it must re-download your repository files.

**b. Log in to Docker Hub**

- name: Log in to Docker Hub

uses: docker/login-action@v3

with:

username: ${{ secrets.DOCKERHUB\_USERNAME }}

password: ${{ secrets.DOCKERHUB\_TOKEN }}

This securely logs in to Docker Hub using **GitHub Secrets** (environment variables stored securely in your repository settings).  
It prevents your credentials from being exposed publicly.

**c. Build and Push Docker Image**

- name: Build and push Docker image

uses: docker/build-push-action@v5

with:

context: .

push: true

tags: ${{ secrets.DOCKERHUB\_USERNAME }}/python-ci-demo:latest

This step:

1. **Builds a Docker image** using your project’s Dockerfile.
2. **Pushes** it to Docker Hub under your account.
3. Tags it as python-ci-demo:latest.

Once this step finishes, your updated application image is live and can be pulled using:

docker pull <your-username>/python-ci-demo:latest

**Summary**

| **Stage** | **Purpose** | **Key Tool** | **Outcome** |
| --- | --- | --- | --- |
| Build | Run tests, verify code | pytest, actions/setup-python | Ensures your app works correctly |
| Deploy | Publish Docker image | docker/build-push-action | Makes your app available on Docker Hub |

After pushing this workflow file, you’ll see two jobs in GitHub Actions:

1. **Build & Test**
2. **Deploy to Docker Hub**

If both go green ✅, check your Docker Hub — you’ll see your new image there!

**Step 5: Verify Deployment**

On Docker Hub, you should see:

python-ci-demo:latest

You can pull and run it anywhere:

docker pull yourusername/python-ci-demo:latest

docker run -p 5000:5000 yourusername/python-ci-demo:latest

docker run -it parteekbhatia954/python-ci-demo:latest python3 -c "from app import add; print(add(2, 3))"

Your app is live locally from Docker Hub!

**Reflection Questions**

1. Explain the difference between Continuous Integration (CI) and Continuous Deployment (CD). How do these two practices work together to ensure reliable and efficient software delivery?
2. What is the purpose of the “checkout” step in a GitHub Actions workflow?
3. What are GitHub-hosted runners, and how do they contribute to automation?
4. What challenges did you encounter while setting up or debugging your workflow,  
   and what specific steps did you take to identify and resolve them?
5. How can you apply the skills and concepts learned from this CI/CD lab  
   to your Software Engineering class project?