Capstone Lab: End-to-End Automated Deployment Pipeline

In this capstone project, you will design and implement a complete end-to-end automated deployment pipeline that demonstrates mastery of modern CI/CD practices. You'll build a scalable, secure, and monitored web application infrastructure using Infrastructure as Code, containerization, and CI/CD pipelines.

This capstone integrates multiple technologies and best practices to create a production-ready system that can automatically deploy.

In this capstone, you will:

- Design and implement Infrastructure as Code using Ansible
- Containerize applications using Docker and orchestrate with Docker Compose
- Build comprehensive CI/CD pipelines with automated testing and deployment
- Implement security best practices and compliance checking
- Demonstrate knowledge of modern deployment strategies

Estimated completion time

130 minutes

Task 1: Project Setup and Infrastructure Design

In this task, you will set up the capstone project structure, design the infrastructure architecture, and create the foundational configuration files.

Step 1: Create the capstone project structure

Create a folder named Capstone-CICD on the Desktop. Open Capstone-CICD in with Visual Studio Code.

Step 2: Initialize project structure

Open VSCode terminal (Terminal → New Terminal) and create the complete project structure:

App

```
mkdir -p application/backend/app application/backend/tests
```

Ansible

```
mkdir -p infrastructure/ansible/{inventories,group_vars,playbooks}
```

Helper script for docker install (copy from Github Docker repo)

```
curl -fsSL https://get.docker.com -o get-docker.sh
cp ~/get-docker.sh infrastructure/ansible/get-docker.sh 2>/dev/null || true
```

Workflows

```
mkdir -p .github/workflows
```

Step 3: Python virtual environment & app code

Create the Vitual Environment:

```
cd ~/Desktop/Capstone-CICD/application/backend
python3 -m venv .venv
source .venv/bin/activate
```

Create the requirements file:

application/backend/requirements.txt

```
Flask==2.3.3
gunicorn==21.2.0
pytest==7.4.3
pytest-html==3.2.0
pytest-cov==4.1.0
```

Create the application code file:

application/backend/app/app.py

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.get("/")
def home():
    return "Hello from Capstone CI/CD!", 200

@app.get("/api/health")
def health():
    return jsonify({"status": "healthy", "version": "latest"}), 200

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)
```

Testing the app:

Run the following command to install all testing dependencies:

```
pip install pytest pytest-cov pytest-html flask
```

This will install:
pytest → testing framework
pytest-cov → for coverage reports
pytest-html → for generating HTML reports

Create the test code files:

application/backend/tests/conftest.py

- This is common in Flask/Django/WSGI entry points.
- It guarantees that when the app runs (especially in Docker, Gunicorn, or other environments), Python knows where your app package is, even if the current working directory is different.

```
import os, sys
# Ensure we can import the `app` package
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
sys.path.insert(0, BASE_DIR)
```

application/backend/tests/test_app.py

```
from app.app import app

def test_home():
    client = app.test_client()
    rv = client.get("/")
    assert rv.status_code == 200
    assert b"Hello" in rv.data

def test_health():
    client = app.test_client()
    rv = client.get("/api/health")
    assert rv.status_code == 200
```

```
    data = rv.get_json()
    assert data["status"] == "healthy"
```

Run tests locally:

```
cd ~/Desktop/Capstone-CICD/application/backend
source .venv/bin/activate
pytest -v --maxfail=1 --disable-warnings \
  --html=report.html --self-contained-html \
  --cov=app --cov-report=html
```

You should see 2 tests passing. If you get ModuleNotFoundError: No module named 'app', the conftest.py above fixes it.

```
(.venv) student@labuser-virtual-machine:~/Desktop/Capstone/application/backend$ pytest -
v --maxfail=1 --disable-warnings   --html=report.html --self-contained-html   --cov=app
--cov-report=html
================================== test session starts ==================================
platform linux -- Python 3.10.12, pytest-8.4.2, pluggy-1.6.0 -- /home/student/Desktop/Ca
pstone/application/backend/.venv/bin/python3
cachedir: .pytest_cache
metadata: {'Python': '3.10.12', 'Platform': 'Linux-6.5.0-44-generic-x86_64-with-glibc2.3
5', 'Packages': {'pytest': '8.4.2', 'pluggy': '1.6.0'}, 'Plugins': {'metadata': '3.1.1',
 'html': '4.1.1', 'cov': '7.0.0'}}
rootdir: /home/student/Desktop/Capstone/application/backend
plugins: metadata-3.1.1, html-4.1.1, cov-7.0.0
collected 2 items

tests/test_app.py::test_home PASSED                                          [ 50%]
tests/test_app.py::test_health PASSED                                        [100%]

================================== tests coverage ==================================
_____________________ coverage: platform linux, python 3.10.12-final-0 _____________________
```

Task 2: Dockerize the application

Step 1: *Create Docker configuration File*

application/backend/Dockerfile

```
# Production image
FROM python:3.11-slim

ENV PYTHONDONTWRITEBYTECODE=1 \
    PYTHONUNBUFFERED=1

WORKDIR /app

# Install system deps (curl helps healthcheck; optional)
RUN apt-get update && apt-get install -y --no-install-recommends curl && \
    rm -rf /var/lib/apt/lists/*

# Copy and install
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy app code
COPY app/ ./app/

EXPOSE 5000

# Healthcheck (tries /api/health)
HEALTHCHECK --interval=30s --timeout=5s --retries=3 \
  CMD curl -fsS http://127.0.0.1:5000/api/health || exit 1

# Gunicorn entrypoint
CMD ["gunicorn", "--bind", "0.0.0.0:5000", "app.app:app", "--workers", "2", "--timeout", "30"]
```

Step 7: *Build and run locally*

cd ~/Desktop/Capstone-CICD/application/backend

docker build -t capstone-flask:local .

```
=> [1/6] FROM docker.io/library/python:3.11-slim@sha256:9bffe4353b925a1656688797  4.3s
=> => resolve docker.io/library/python:3.11-slim@sha256:9bffe4353b925a1656688797  0.0s
=> => sha256:44350d10c02e7ab437e3fe5a05e3405115ece5972b2b9f7cd0d 1.29MB / 1.29MB  3.3s
=> => sha256:4dc2c3222cdbf7b5e9d5c68653d42c7289ddf2bfaa17b12c9 14.64MB / 14.64MB  1.2s
=> => sha256:b25238518c0cca0928b2117b90cee455c3fbdb7d605f92131e5cc92 249B / 249B  1.4s
=> => sha256:9bffe4353b925a1656688797ebc68f9c525e79b1d377a764d 10.37kB / 10.37kB  0.0s
=> => sha256:70f7abeaf1577b30229dd1d7784d6c053a29104a56bb353fe23 1.75kB / 1.75kB  0.0s
=> => sha256:bf02a2b853727373d9065ccd2cc7d40df56d6f1b8256ae5f361 5.38kB / 5.38kB  0.0s
=> => extracting sha256:44350d10c02e7ab437e3fe5a05e3405115ece5972b2b9f7cd0d68d23  0.1s
=> => extracting sha256:4dc2c3222cdbf7b5e9d5c68653d42c7289ddf2bfaa17b12c96101475  0.9s
=> => extracting sha256:b25238518c0cca0928b2117b90cee455c3fbdb7d605f92131e5cc92f  0.0s
=> [internal] load build context                                                  0.0s
=> => transferring context: 1.24kB                                                0.0s
=> [2/6] WORKDIR /app                                                             0.1s
=> [3/6] RUN apt-get update && apt-get install -y --no-install-recommends curl   27.0s
=> [4/6] COPY requirements.txt .                                                  0.0s
=> [5/6] RUN pip install --no-cache-dir -r requirements.txt                      20.3s
=> [6/6] COPY app/ ./app/                                                         0.0s
=> exporting to image                                                             0.2s
=> => exporting layers                                                            0.2s
=> => writing image sha256:6e3066a50e9dbe8b430cd7f0a09d85f5ae9306d93b109beb7235c  0.0s
=> => naming to docker.io/library/capstone-flask:local                            0.0s
```

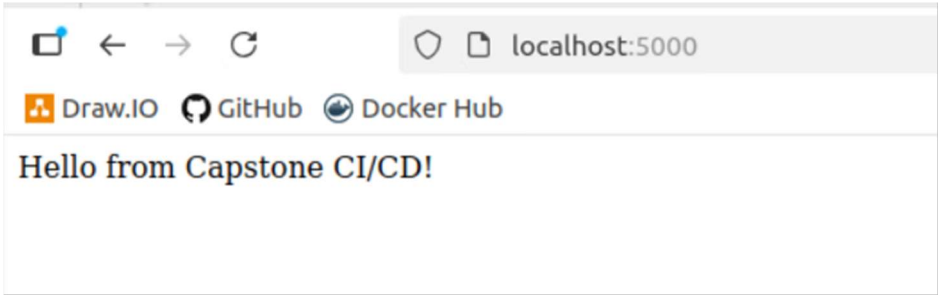
Docker login ??

docker run --rm -d -p 5000:5000 --name capstone_flask capstone-flask:local

```
(.venv) student@labuser-virtual-machine:~/Desktop/Capstone/application/backend$ docker r
un --rm -d -p 5000:5000 --name capstone_flask capstone-flask:local
bb689d117fc3cb358cc0e69c0002d0253b52fd386d31df8a1204604b8b1d3a2f
(.venv) student@labuser-virtual-machine:~/Desktop/Capstone/application/backend$
```

curl -fsS http://localhost:5000/api/health

```
(.venv) student@labuser-virtual-machine:~/Desktop/Capstone/application/backend$ curl -fs
S http://localhost:5000/api/health
{"status":"healthy","version":"latest"}
(.venv) student@labuser-virtual-machine:~/Desktop/Capstone/application/backend$
```



docker stop capstone_flask

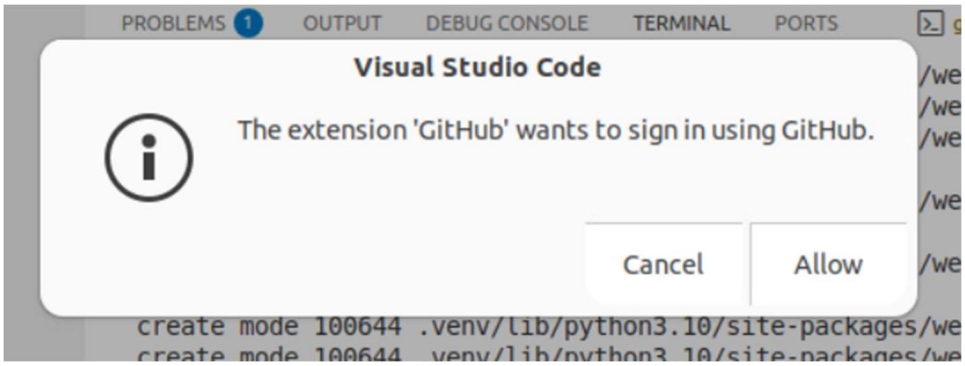
Task 3: Initialize Git and push to GitHub

First, create a new repo on GitHub named **capstone-cicd** (empty i.e. no README).

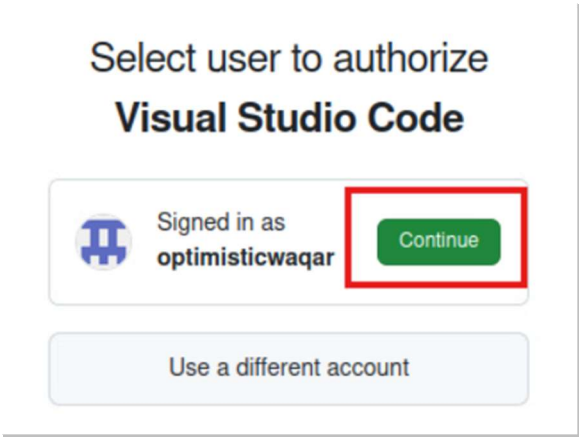
Then:

```
cd ~/Desktop/Capstone-CICD
git init
git branch -M main
git remote add origin https://github.com/<YOUR_USER>/capstone-cicd.git
git add .
git commit -m "Initial commit: app, tests, Dockerfile, ansible skeleton, workflows scaffolding"
git push -u origin main
```

Allow the extension 'Github' to sign in to Github.



Click continue:



Task 4: GHCR (GitHub Container Registry) setup

You have two choices:

- **Public image (easiest):** No auth required to pull
- **Private image: Requires login** (GHCR_USER, GHCR_PAT)

(A) Make image Public

Do nothing now; the workflow tags will create it, and you can change visibility to Public from the package page later.

(B) Keep image Private

Create a **Personal Access Token (classic)** with scopes:

- read:packages, write:packages, (optional delete:packages), repo

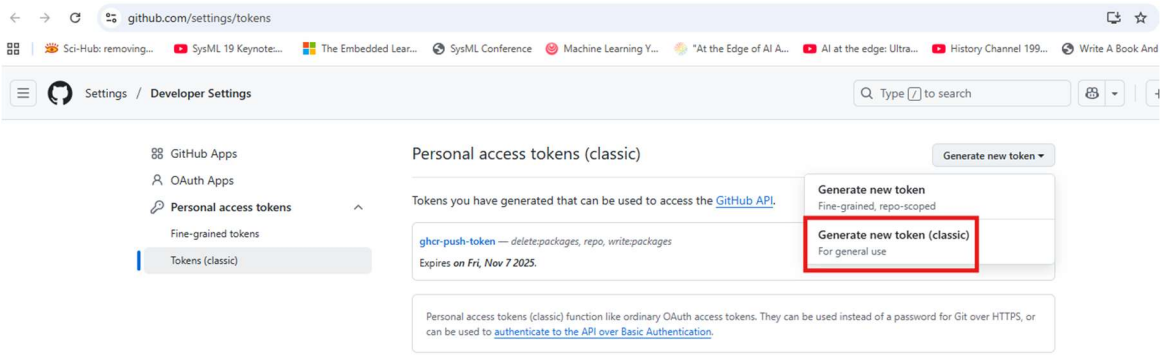
Then add repo **Secrets** (GitHub → Repo → Settings → Secrets and variables → Actions):

- GHCR_USER = your GitHub username (e.g., optimisticwaqar)
- GHCR_PAT = your personal access token value

Steps to Create a GHCR PAT

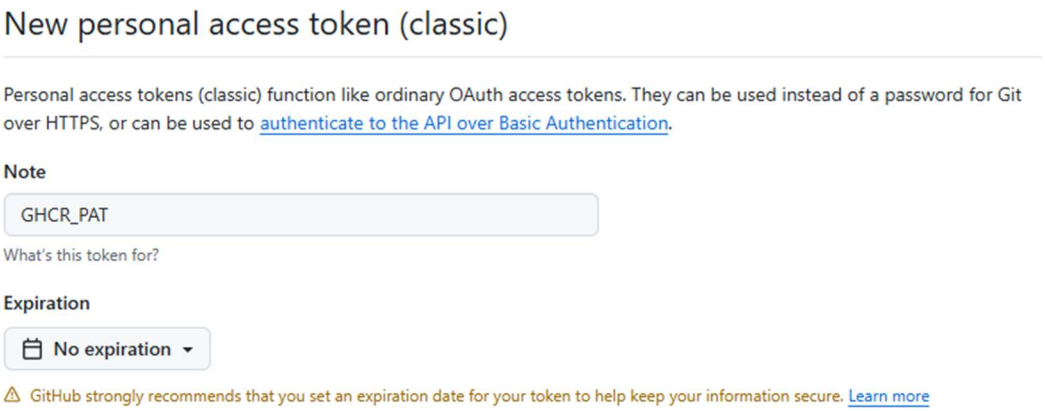
Step 1: Go to your GitHub Settings

- Open https://github.com/settings/tokens.
- Click **"Generate new token" → "Generate new token (classic)".**



Step 2: Give the Token a Name & Expiration

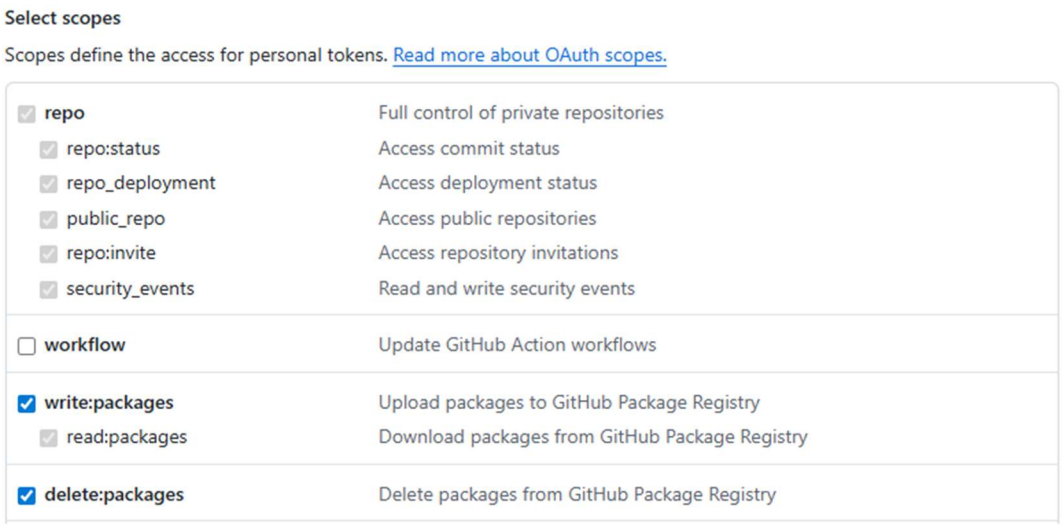
- Add a descriptive name like **GHCR_PAT**.
- Set an expiration date (or "No expiration" if needed for CI/CD).



Step 3: Select Required Scopes

For GHCR, you typically need:

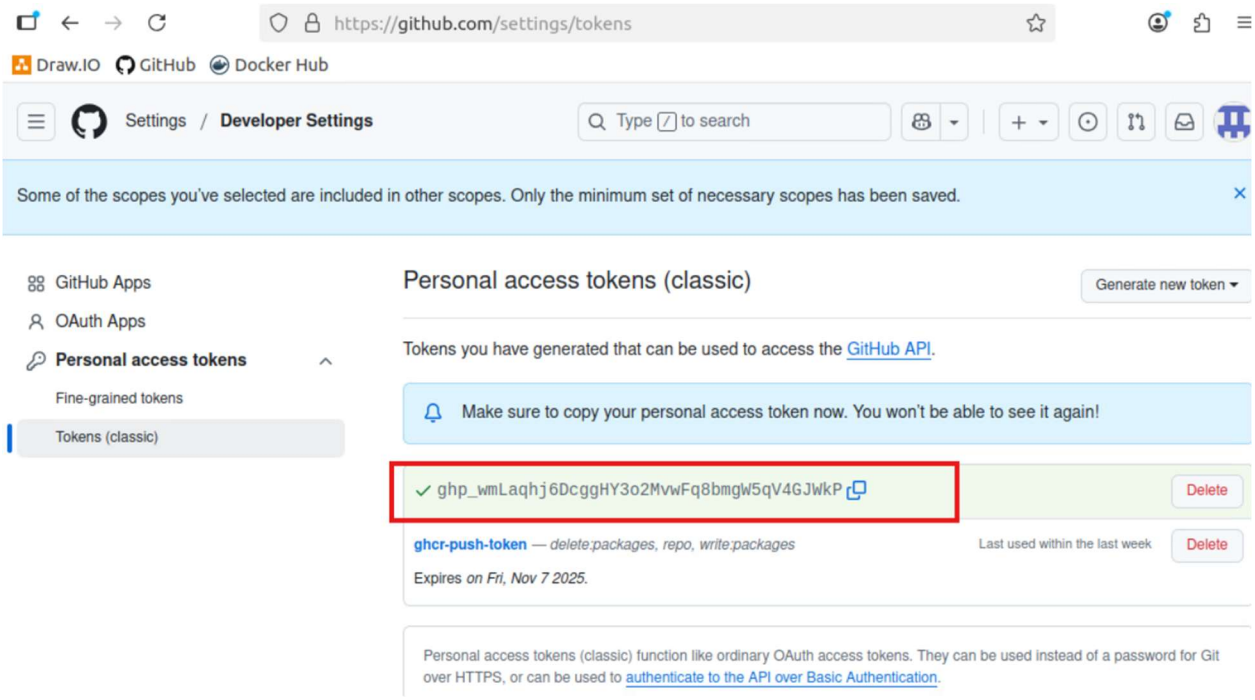
- read:packages → if only pulling images
- write:packages → if pushing images
- delete:packages → if you want to delete packages
- repo → if the package is private and linked to a repo



For CI/CD pushing: **check** write:packages **and** read:packages (and repo if your repo is private).

Step 4: Generate the Token

- Click Generate token.
- Copy it immediately (you can't see it later).



Task 5: Ansible: inventory, vars, deploy playbook

Step 1: Create development.yml

infrastructure/ansible/inventories/development.yml

```
---
all:
  hosts:
    localhost:
      ansible_connection: local
      ansible_python_interpreter: /usr/bin/python3
  vars:
    app_port: 5000
    image_ref: "ghcr.io/<YOUR_USER>/capstone-cicd/capstone-flask:latest"
    ghcr_private: false   # set true if your GHCR package is private
```

> **Note: Replace <YOUR_USER> with your GitHub username.**

Step 2: Create all.yml

infrastructure/ansible/group_vars/all.yml

```
---
app_port: 5000
image_ref: "ghcr.io/<YOUR_USER>/capstone-cicd/capstone-flask:latest"
ghcr_private: false
```

> **Note: Replace <YOUR_USER> with your GitHub username.**

Step 3: Create deploy-app.yml

infrastructure/ansible/playbooks/deploy-app.yml

```
---
- name: Deploy Flask App using Docker
  hosts: all
  become: yes

  vars:
    image: "{{ image_ref }}"

  tasks:
    - name: Check if docker exists
      command: bash -lc "command -v docker || true"
      register: docker_cmd
      changed_when: false

    - name: Install Docker via script if missing
      shell: |
        set -e
        chmod +x {{ playbook_dir }}/../get-docker.sh || true
        sudo sh {{ playbook_dir }}/../get-docker.sh
      when: docker_cmd.stdout == ""
      args:
        executable: /bin/bash

    - name: Ensure docker service is started
      service:
        name: docker
        state: started
        enabled: yes

    - name: Verify Docker service
      shell: systemctl is-active docker
```

```
    register: docker_status
    changed_when: false
    failed_when: docker_status.stdout.strip() != "active"

  - name: Log in to GHCR (only if private)
    command: >
      docker login ghcr.io
      -u {{ lookup('env','GHCR_USER') }}
      --password {{ lookup('env','GHCR_PAT') }}
    register: ghcr_login
    changed_when: "'Login Succeeded' in ghcr_login.stdout"
    retries: 3
    delay: 5
    until: ghcr_login.rc == 0
    when: ghcr_private | bool

  - name: Pull Flask application image
    command: docker pull {{ image }}
    register: pull_result
    retries: 3
    delay: 5
    until: pull_result.rc == 0
    ignore_errors: no

  - name: Stop existing container if running
    command: docker stop capstone_flask
    ignore_errors: yes

  - name: Remove old container if exists
    command: docker rm capstone_flask
    ignore_errors: yes

  - name: Run application container
```

```
    command: >
      docker run -d --name capstone_flask
      -p {{ app_port }}:5000
      {{ image }}

  - name: Wait for Flask application to start
    uri:
      url: "http://localhost:{{ app_port }}/api/health"
      method: GET
      status_code: 200
    register: app_status
    retries: 15
    delay: 4
    until: app_status.status == 200

  - name: Display success message
    debug:
      msg: "☑ Flask application deployed successfully and is healthy!"
```

This playbook installs Docker (only if missing), optionally logs in to GHCR if your package is private, pulls the image, and runs it.

Task 6: GitHub Actions: CI (build, test, push) & Deploy

In this final task, you will integrate all components, run comprehensive tests, and validate the complete DevOps pipeline.

Step 1: *Builds, tests, and pushes the Docker image to GHCR using GITHUB_TOKEN*

.github/workflows/ci.yml

```
name: DevOps Capstone - Continuous Integration

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]
  workflow_dispatch: {}

env:
  IMAGE: ghcr.io/${{ github.repository }}/capstone-flask
  PYTHON_VERSION: '3.11'

jobs:
  build-test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - uses: actions/setup-python@v4
        with:
          python-version: ${{ env.PYTHON_VERSION }}

      - name: Install deps & run tests
        working-directory: application/backend
```

```
      run: |
        python -m pip install --upgrade pip
        pip install -r requirements.txt
        pytest -v --maxfail=1 --disable-warnings \
          --html=report.html --self-contained-html \
          --cov=app --cov-report=xml --cov-report=html

    - name: Upload test artifacts
      uses: actions/upload-artifact@v4
      with:
        name: test-report
        path: application/backend/report.html

  docker-build-push:
    runs-on: ubuntu-latest
    needs: build-test
    permissions:
      contents: read
      packages: write
    steps:
      - uses: actions/checkout@v4

      - name: Log in to GHCR
        uses: docker/login-action@v3
        with:
          registry: ghcr.io
          username: ${{ github.actor }}
          password: ${{ secrets.GITHUB_TOKEN }}

      - name: Set up Buildx
        uses: docker/setup-buildx-action@v3

      - name: Build & push
```

```
      uses: docker/build-push-action@v5
      with:
        context: application/backend
        push: true
        tags: |
          ${{ env.IMAGE }}:latest
          ${{ env.IMAGE }}:${{ github.sha }}
```

> **Note:** This uses GITHUB_TOKEN to push to GHCR for this repository namespace. If your package is private and later pulled by Ansible on the same runner, add a login step in deploy workflow or playbook (we added in playbook already, controlled by ghcr_private).

Step 2: *Runs Ansible to deploy (either automatically after CI or manually)*

.github/workflows/deploy.yml

```
name: DevOps Capstone - Deployment

on:
 workflow_run:
  workflows: ["DevOps Capstone - Continuous Integration"]
  types: [completed]
  branches: [ main ]
 workflow_dispatch:
  inputs:
   ghcr_private:
    description: "Is GHCR package private?"
    required: true
    type: choice
    options: [ "false", "true" ]
    default: "false"

jobs:
 deploy:
```

```
  runs-on: ubuntu-latest
  permissions:
    contents: read
    packages: read
  env:
    GHCR_USER: ${{ secrets.GHCR_USER }}
    GHCR_PAT:  ${{ secrets.GHCR_PAT }}
  steps:
    - name: Checkout
      uses: actions/checkout@v4

    - name: Install Ansible
      run: |
        python3 -m pip install --upgrade pip
        pip3 install ansible

    - name: Render inventory with chosen visibility
      run: |
        sed -i "s/ghcr_private: .*/ghcr_private: ${{ github.event.inputs.ghcr_private || 'false' }}/" infrastructure/ansible/group_vars/all.yml || true

    - name: Run deployment (local deploy on runner)
      working-directory: infrastructure/ansible
      run: |
        ansible-playbook -i inventories/development.yml playbooks/deploy-app.yml

    - name: Smoke test
      run: |
        curl -fsS http://localhost:5000/api/health
```

What this does:

- If CI succeeded on main, this deploys automatically.
- You can also Run workflow manually and choose whether GHCR is private.
- It deploys to the GitHub runner (ephemeral) for demo.

Task 7: Full Run

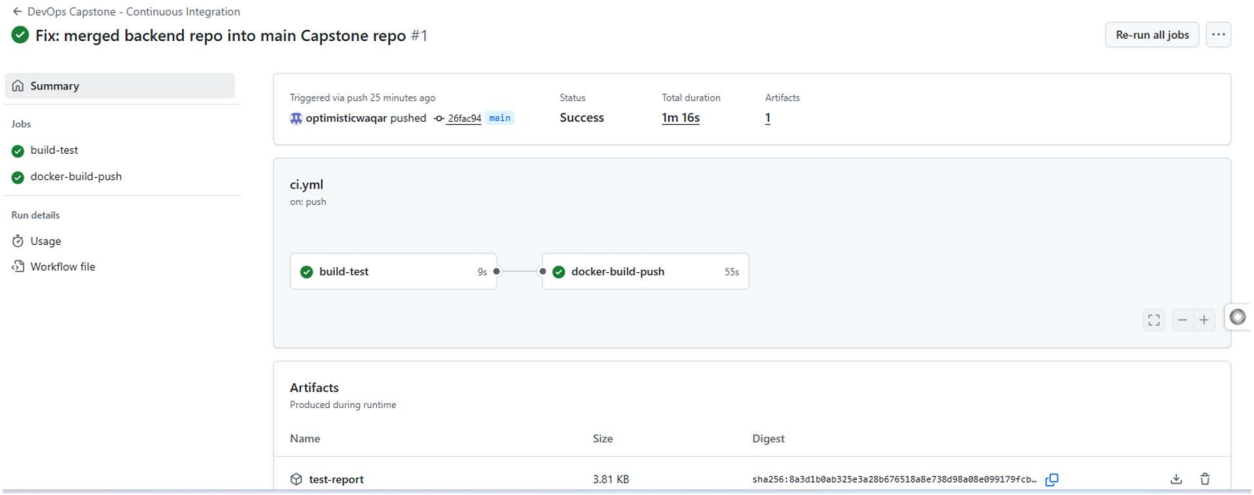
In this final task, you will integrate all components, run comprehensive tests, and validate the complete DevOps pipeline.

Step 1: *Commit & push:*

```
cd ~/Desktop/Capstone-CICD
git add .
git commit -m "CI, Docker, Ansible deploy wired up"
git push
```

Step 2: *Watch Actions tab:*

- CI builds/tests & pushes image
- Deploy runs Ansible, pulls image, starts container, checks /api/health



Capstone Lab Summary: End-to-End CI/CD Pipeline

The objective of this lab was to implement a **complete CI/CD pipeline** that automatically builds, tests, packages, and deploys a Flask web application using modern DevOps tools — **GitHub Actions, Docker, and Ansible** — while following industry best practices for automation and reliability.

Theoretical Flow and Key Concepts

1. Continuous Integration (CI)

Continuous Integration automates the process of building and testing code whenever developers push updates to the main repository.
In this lab:

- GitHub Actions automatically triggered the CI workflow (ci.yml) when code was pushed or when a pull request was created on the main branch.
- The workflow:
- Checked out the repository.
- Installed dependencies in a Python environment.
- Ran Pytest to execute automated unit tests (test_app.py).
- Generated test coverage reports (HTML and XML).
- Built and pushed a Docker image to GitHub Container Registry (GHCR) tagged as latest and by commit.

This ensured every commit was tested and built consistently, guaranteeing the application is always in a deployable state.

---

2. Continuous Delivery (CD)

Continuous Delivery focuses on automatically deploying applications once they have passed the integration and testing stages.

In this lab:

The deployment workflow (deploy.yml) ran automatically after the CI pipeline completed successfully.

The workflow:

- Installed Ansible in the GitHub runner.
- Used the Ansible playbook (deploy-app.yml) to deploy the containerized application.

- Pulled the Docker image from GHCR.
- Stopped and removed old containers (if any).
- Started a new Docker container using the latest image.
- Performed a health check using the /api/health endpoint.

This automated flow demonstrates how code moves seamlessly from source control → containerization → deployment without manual intervention.

---

3. Infrastructure as Code (IaC) with Ansible

Ansible was used to define infrastructure configuration and deployment logic declaratively:

- Inventories defined target hosts (development.yml for local deployment).
- Group variables stored reusable app configuration (all.yml).
- Playbooks described each deployment step in YAML, making it reproducible and easy to maintain.
- The playbook also ensured Docker was installed and running before deploying the app.

By using Ansible, deployment became idempotent (safe to re-run multiple times) and platform-independent.

---

4. GitHub Container Registry (GHCR)

The Docker image was stored in GHCR, serving as a secure and versioned artifact repository. This ensured that deployments always used a known image version, preventing “works on my machine” issues.

Access was managed via GitHub secrets (GHCR_USER, GHCR_PAT) for authentication, making the workflow both secure and automated.

---

5. Workflow Orchestration

The pipeline demonstrated two interconnected workflows:

1. CI Pipeline (ci.yml)

- Triggered by a code push or pull request.
- Built and tested the application.
- Pushed the Docker image to GHCR.

2. CD Pipeline (deploy.yml)

- Triggered automatically when the CI pipeline succeeded.
- Used Ansible to deploy the app in a container.

This orchestration implemented the Continuous Deployment concept: deploy automatically after successful testing.

---

6. Automation, Reproducibility, and Visibility

- Every stage was automated (no manual setup required).
- Each workflow's logs were visible in the GitHub Actions tab.
- If something failed (e.g., Docker login, missing image, or health check), the workflow clearly indicated the reason for failure.
- Using GitHub-hosted runners kept the setup lightweight and reproducible.

Lab review

In the CI/CD pipeline implemented, what triggers automated deployment to production?

A. Every commit to any branch
B. Manual approval after successful staging deployment
C. Automatic deployment on main branch
D. Only scheduled deployments

Correct Answer: B. Manual approval after successful staging deployment

STOP

You have successfully completed this lab.