GitHub Actions Pipeline Creation Assignment

Intern: Sumit Dhar
Project: Calculator Application (using GitHub Actions)
Date: 10th September, 2025

---

1. Introduction

This report documents the implementation of a **CI pipeline using GitHub Actions** for a Python-based sample application. The pipeline automates code checkout, dependency installation, linting, unit testing, and artifact upload, while being easily extensible for containerization or cloud deployment

Key Objectives:

- Learn Git, GitHub, DevOps culture and CI/CD best practices.
- Automate builds and tests using GitHub Actions workflows.
- Ensure code quality through linting and unit testing.
- Speed up execution using caching mechanisms.
- Provide instant feedback to developers with email notifications.
- Prepare the pipeline for future extensions (Docker builds, AWS/GCP deployment).

---

2. Pre-requisites

Before setup, the following tools and configurations were required:

1. **System & Tools**
   - Windows/Linux/Mac system with internet access
   - GitHub account and repository (this project: `dhar-sumit/demo-repo2`)
2. **Installed Tools (Local)**
   - Git installed and configured
   - Python (3.11+ recommended) with `pip`
   - Text editor/IDE (e.g., VS Code, PyCharm)
3. **Knowledge Requirements**
   - Basic familiarity with Git commands (clone, commit, push, pull)
   - Branching workflow (feature branch → PR → merge to `main`)
4. **GitHub Setup**
   - Repository structured with `src/` and `tests/` directories
   - `.github/workflows/` directory to hold workflow files
   - GitHub Actions enabled (default for repositories)

---

3. Environmental Setup

Step 1: Create Repository

- Log in to GitHub and create a repository (e.g., `demo-repo2`).
- Initialize with a `README.md`, `.gitignore`, and license if required.

Step 2: Project Structure

```
demo-repo2/
├─ src/
│   └─ calc.py
├─ tests/
│   └─ test_calc.py
├─ requirements.txt
├─ .gitignore
├─ README.md
└─ .github/
    └─ workflows/
        └─ main.yml
```

Notes:

- `requirements.txt` contains `pytest` and `flake8` (and other runtime libs if needed).
- Write the codes in src/ and tests/ and keep tests fast and isolated to maintain quick CI feedback loops.

Step 3: Enable GitHub Actions

- Go to repository → **Actions tab**.
- GitHub will auto-detect workflows if `.github/workflows/` exists.

Step 4: Configure Python & Dependencies

- Use `actions/setup-python@v5` to install Python versions (3.11, 3.12, 3.13).
- Install dependencies via `pip install -r requirements.txt`.

Step 5: Add Workflow File

- Create `.github/workflows/main.yml`.
- Define triggers, jobs, and steps (checkout, setup Python, lint, test).

Step 6: Notifications & Caching

- Use `actions/cache@v3` to cache `pip` dependencies for faster builds.
- Enable email notifications (default by GitHub).

Step 7: Verify Setup

- Commit and push changes to `main`.
- Workflow will trigger automatically on push/PR.
- Status can be seen under the **Actions tab** and in pull requests

---

4. GitHub Actions Workflow (Pipeline Stages)

Pipeline Stages:

1. **Checkout** → Pull the latest code from the GitHub repository.
2. **Set up Python** → Configure multiple Python versions (3.11, 3.12, 3.13) using `actions/setup-python`.
3. **Cache Dependencies** → Use `actions/cache` to store and restore pip dependencies for faster builds.
4. **Install Dependencies** → Upgrade pip and install all required packages from `requirements.txt`.
5. **Lint** → Run `flake8` to ensure code style and quality.
6. **Run Tests** → Execute `pytest`, generate JUnit XML reports, and validate correctness.
7. **Upload Artifacts** → Store test reports as downloadable artifacts for each Python version.
8. **Notifications** → Email alerts are triggered automatically on workflow success or failure.

Pipeline Configuration (main.yml key extract):

```
jobs:
  build-and-test:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        python-version: [3.11, 3.12, 3.13]

    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Set up Python
        uses: actions/setup-python@v5
        with:
          python-version: ${{ matrix.python-version }}
          cache: 'pip'

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

      - name: Lint (flake8)
        run: flake8 src tests

      - name: Run tests (pytest)
        run: pytest --junitxml=reports/junit-${{ matrix.python-version }}.xml
```

5. Observability & Artifacts

Artifacts:

- Test results (`pytest` reports) are uploaded as workflow artifacts for download and review.
- Build logs remain accessible from the GitHub Actions dashboard for debugging.

Caching:

- Python dependencies (`pip`) are cached using `actions/cache`, reducing redundant downloads and improving build speed.

Visibility:

- Contributors can track build status directly on the **Actions tab** or through email notifications.
- Failed runs highlight errors with logs linked to the exact step.

Key Benefit:
Provides transparency, faster debugging, and efficient builds by avoiding repeated work.

---

6. Local testing (how to reproduce & test)

```
# create & activate virtualenv
python -m venv <venv_name>
source .venv/bin/activate

# install deps
pip install -r requirements.txt

# run lint locally (see issues)
flake8 src tests

# run tests locally
pytest -q

# to generate same junit report locally:
pytest --junitxml=reports/junit.xml
```

---

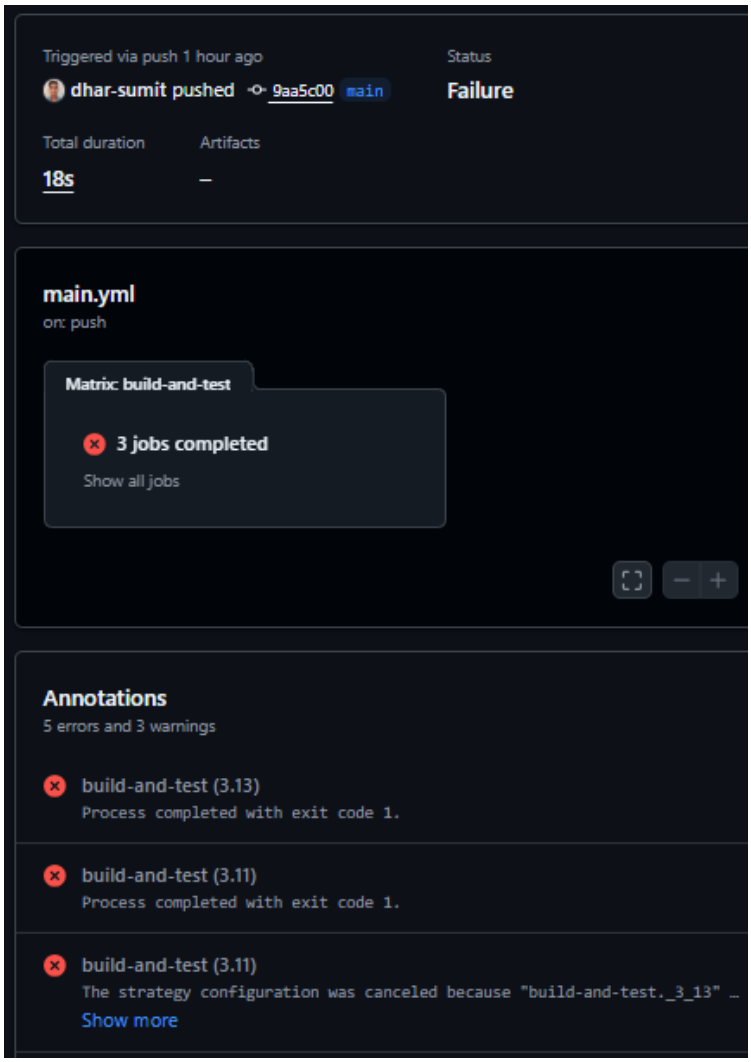
7. How to test notifications & caching safely

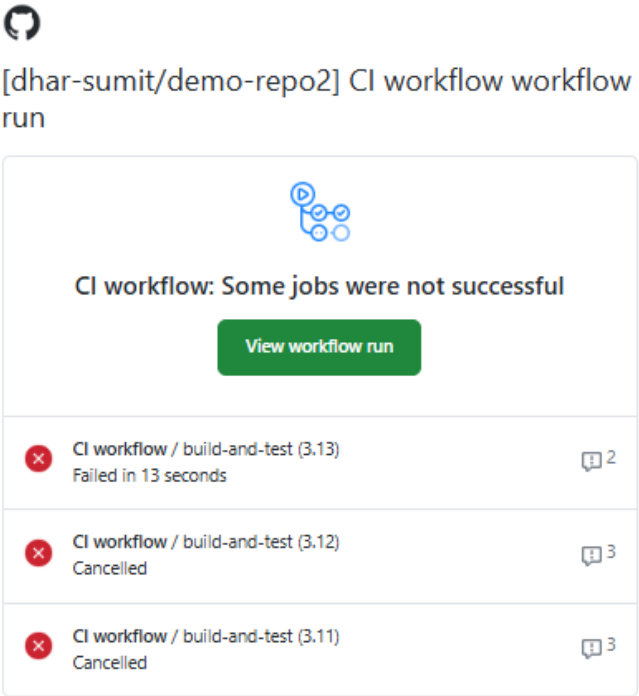
Test caching:

- Push a commit that only touches documentation (no dependency changes) and observe that `pip install` is faster due to cache.

Test failure notification (safe):

- Create a temporary failing test in a branch (e.g., `assert False`) and push to a test branch you allowed in triggers. The `notify` job will send an email to `NOTIFY_EMAIL` if the build fails.
- Remove the intentional failure after verifying email receipt.





Triggered via push 1 hour ago

dhar-sumit pushed 3c35700 main

Status

Success

Total duration

20s

Artifacts

3

main.yml

on: push

Matrix: build-and-test

- build-and-test (3.11) 14s
- build-and-test (3.12) 13s
- build-and-test (3.13) 15s

Artifacts

Produced during runtime

Name	Size	Digest
junit-report-3.11	392 Bytes	sha256:91f55d692...
junit-report-3.12	391 Bytes	sha256:f51ee4f8c...
junit-report-3.13	392 Bytes	sha256:8eff2decb...

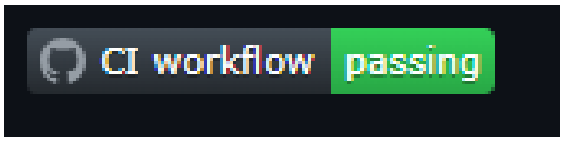
8. Key Learnings & Observations

• **Automation** → Reduced manual effort by running tests and builds automatically on each push.
• **CI with GitHub Actions** → Reliable and consistent pipeline directly integrated with GitHub.
• **Efficiency** → Caching pip dependencies sped up workflow execution.
• **Faster Feedback** → Developers get immediate notifications and build/test results.
• **Documentation** → Clear README instructions improved collaboration and reproducibility.

---

9. Challenges encountered

- Understanding badge caching — badge can be delayed ~1–2 minutes.
- Deciding between Slack and Email for notifications — Email chosen to avoid external workspace setup.
- Initial flake8 failures blocked workflow until tests were stabilized.



---

10. Conclusion

This assignment successfully demonstrated setting up a CI/CD pipeline for a Python-based sample repository using **GitHub Actions**.
✓ Local development + unit testing
✓ GitHub version control
✓ GitHub Actions pipeline automation
✓ Dependency caching for faster builds
✓ Email notifications for workflow status
