

DevOps Homework 3 - Winter 2026

Student: César Núñez

Date: 01/26/2026

Question 1

Compare and contrast at least three CI tools/platforms (e.g., Jenkins, GitHub Actions, GitLab CI, Azure DevOps, CircleCI). Create a comparison table that includes:

- Pricing model
- Ease of setup
- Integration capabilities
- Key features
- Best use cases

CI Tool/Platform	Pricing model	Ease of setup	Integration capabilities	Key features	Best case uses
Jenkins	Free and Open Source, but you need to pay for the machine it runs on	Setup is harder since you have to install Jenkins, manage plug ins (ex. Docker) which can take a lot of time	It has a lot of plug-ins that allows the integration with different services (GitHub, AWS, Docker, etc.)	Customizable pipelines, full control over the pipeline	When you need full control of the pipeline/infrastructure and custom behavior
GitHub Actions	Free for public repositories. Private repositories have a monthly allowance of build minutes, then pay per minute.	Easier. Just need to create a YAML file that will run CI automatically	Integration with GitHub features (PR, issues and releases)	Automatic checks for other GitHub features, YAML-based workflows	Small-medium projects hosted in GitHub

CI Tool/Platform	Pricing model	Ease of setup	Integration capabilities	Key features	Best case uses
Azure DevOps	<p>Free for individuals up to 1800 minutes per month in one Microsoft-hosted jobs and unlimited minutes for 1 self-hosted job.</p> <p>\$40 per extra Microsoft-hosted CI/CD parallel job</p>	<p>More complex relative to GitHub Actions/Jenkins</p>	<p>Integrations with Azure services and supports external repos</p>	<p>CI/CD pipelines, enterprise grade access control</p>	<p>Better for enterprises that use other Azure services</p>

Question 2

1. CI Pipeline Configuration

- Complete pipeline configuration file (Jenkinsfile, .github/workflows/ci.yml, etc.): available [here](#) and also here:

```

name: ci_ass3

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

permissions:
  contents: read

jobs:
  build:
    runs-on: ubuntu-latest

    env:
      COMPOSE_PROJECT_NAME: ${{ github.workflow }}_${{ github.run_id }}_${{ github.run_attempt }}

```

```
COVERAGE_FAIL_UNDER: "80"

steps:
  - uses: actions/checkout@v4.2.2

  - name: Create env file
    run: |
      echo "${{ secrets.DEVOPS_ASS3 }}" > .env

  - name: Build images
    run: |
      docker compose -p "$COMPOSE_PROJECT_NAME" build db backend test

  - name: Start database
    run: |
      docker compose -p "$COMPOSE_PROJECT_NAME" up -d db

  - name: Code Quality Checks
    run : |
      docker compose -p "$COMPOSE_PROJECT_NAME" run --rm test \
        ruff check . --fix
      docker compose -p "$COMPOSE_PROJECT_NAME" run --rm test \
        ruff format --check .

  - name: Code Testing
    run : |
      set -eu
      mkdir -p reports

      docker compose -p "$COMPOSE_PROJECT_NAME" run --rm \
        -v "$PWD/reports:/backend/reports" \
        test sh -lc "
          pytest -q \
            --junitxml=/backend/reports/junit.xml \
            --cov=. \
            --cov-report=xml:/backend/reports/coverage.xml \
            --cov-report=html:/backend/reports/htmlcov \
            --cov-fail-under=$COVERAGE_FAIL_UNDER
        "
    - name: Upload test reports
      if: always()
      uses: actions/upload-artifact@v4
      with:
        name: test-reports
        path: reports/

  - name: Shutdown
    if: always()
    run: |
      docker compose -p "$COMPOSE_PROJECT_NAME" down -v
```

- Pipeline must implement all 5 required tasks above

The screenshot shows a CI pipeline build log titled "build" which succeeded 3 minutes ago in 1m 0s. The log details 11 steps, each with a checkmark indicating success:

Step	Description	Duration
> 1	Set up job	1s
> 2	Run actions/checkout@v4.2.2	1s
> 3	Create env file	0s
> 4	Build images	32s
> 5	Start database	12s
> 6	Code Quality Checks	6s
> 7	Code Testing	2s
> 8	Upload test reports	1s
> 9	Shutdown	1s
> 10	Post Run actions/checkout@v4.2.2	0s
> 11	Complete job	0s

2. Application Source Code

- Working application with minimum 3 endpoints/features: available [here](#)
- Unit tests with minimum 5 test cases: available [here](#)
- README file with project setup instructions: available [here](#)
- All necessary configuration files: use this .env file (also in the README.md)

```
MYSQL_ROOT_PASSWORD=secret
MYSQL_DATABASE=todos
MYSQL_USER=todo_user
MYSQL_PASSWORD=todo_pass
```

3. Pipeline Evidence

- Screenshots or links showing successful pipeline runs:
 - Example 1: [click here](#)
 - Example 2: [click here](#)

- Evidence that all 5 pipeline tasks execute successfully

A screenshot of a GitHub Actions build log titled "build". The log shows the following steps:

- > ✓ Set up job 1s
- > ✓ Run actions/checkout@v4.2.2 1s
- > ✓ Create env file 0s
- > ✓ Build images 32s
- > ✓ Start database 12s
- > ✓ Code Quality Checks 6s
- > ✓ Code Testing 2s
- > ✓ Upload test reports 1s
- > ✓ Shutdown 1s
- > ✓ Post Run actions/checkout@v4.2.2 0s
- > ✓ Complete job 0s

- Screenshots showing pipeline failures (e.g., failed tests, code quality issues)

- Coverage fail: [click here](#)

A screenshot of a GitHub Actions log titled "Code Testing". The log shows the following output:

```
1 ► Run set -eu
19 Container ci_ass3_21412257231_1-db-1 Running
20 ..
21 ERROR: Coverage failure: total of 67 is less than fail-under=80
22 [100%]
23 ===== tests coverage =====
24 coverage: platform linux, python 3.11.14-final-0
25
26 Coverage HTML written to dir /backend/reports/htmlcov
27 Coverage XML written to file /backend/reports/coverage.xml
28 FAIL Required test coverage of 80% not reached. Total coverage: 67.31%
29 2 passed in 0.39s
30
31 Error: Process completed with exit code 1.
```

- Test fail: [click here](#)

Code Testing 2s

```
1 ► Run set -eu
19 Container ci_ass3_21414810036_1-db-1 Running
20 ....F [100%]
21 ===== FAILURES =====
22 test_get.todos.sorted_by_deadline
23 client = <FlaskClient <Flask 'app.app'>>
24     def test_get.todos.sorted_by_deadline(client):
25         client.post(
26             "/api/todos",
27             json={"title": "Later", "category": "Work", "deadline": "2026-02-
28             01"},
29         )
30         client.post(
31             "/api/todos",
32             json={"title": "Sooner", "category": "Work", "deadline": "2026-01-
33             15"},
34         )
35         res = client.get("/api/todos")
36     >     assert res.status_code == 0
37 E     assert 200 == 0
38 E     + where 200 = <WrapperTestResponse streamed [200 OK]>.status_code
39 app/tests/test_app.py:109: AssertionError
```

- Code Quality fails: [click here](#)

Code Quality Checks 5s

```
1 ► Run docker compose -p "$COMPOSE_PROJECT_NAME" run --rm test \
10 Container ci_ass3_21412018657_1-db-1 Running
11 F401 [*] `datetime.datetime` imported but unused
12 --> app/tests/test_app.py:4:22
13 |
14 2 | import pytest
15 3 | from lxml.html import fromstring
16 4 | from datetime import datetime
17  |          ^^^^^^^^
18  |
19 help: Remove unused import: `datetime.datetime`
20
21 Found 1 error.
22 [*] 1 fixable with the `--fix` option.
23
24 Error: Process completed with exit code 1.
```

Question 3

Implementing a CI pipeline came with several challenges, especially during the initial setup phase. At the beginning of this homework, I chose Jenkins and was able to run it locally alongside my containerized application. While this helped me understand how CI tools work under the hood, I quickly ran into friction. The Jenkins documentation felt outdated and fragmented, and when I tried to configure GitHub triggers, the lack of clear and modern examples made debugging unnecessarily difficult. Much of my time was spent troubleshooting configuration issues rather than improving the pipeline itself, which made the process frustrating and inefficient.

Because of these challenges, I decided to switch to GitHub Actions. This change significantly improved my experience. The documentation was clear, well-maintained, and filled with practical examples, especially for Python and Docker-based applications (<https://github.com/actions/starter-workflows/blob/main/ci/docker-image.yml>). The availability of starter workflows made it easy to bootstrap a working CI pipeline with minimal configuration. Compared to Jenkins, GitHub Actions felt more intuitive and better integrated with the GitHub ecosystem, allowing me to focus on defining meaningful checks rather than fighting the tool.

Using CI has already changed how I think about software development. Going forward, I plan to include CI pipelines in all my projects from the start. Automatically running code quality checks and unit tests on every commit and push gives me faster feedback and more confidence in my changes. Instead of manually testing or worrying about breaking existing functionality, I can rely on CI to catch issues early. This encourages smaller, more frequent commits and reinforces better development habits overall.

Given more time, one improvement I would make to my pipeline is publishing Docker images to Docker Hub as part of the CI process. This would allow me to easily deploy and run my application in any environment without rebuilding images manually. Additionally, I would like to learn how to better integrate GitHub and Jenkins, as understanding how these tools can complement each other would be valuable for working in more complex or legacy CI/CD environments in the future.