Git and CI/CD

Week 11 · CS 203: Software Tools and Techniques for AI

Prof. Nipun Batra
IIT Gandhinagar

The Problem: Manual Deployment

Without automation:

1. Write code
2. Run tests (maybe?)
3. Commit changes
4. SSH into server
5. Pull latest code
6. Restart the app
7. Hope nothing breaks

With automation (CI/CD):

1. `git push`
2. Tests run automatically
3. Code deploys if tests pass

What is CI/CD?

CI = Continuous Integration

- Merge code frequently (daily)
- Run tests on every push
- Catch bugs early

CD = Continuous Deployment

- Automatically deploy when tests pass
- No manual intervention needed
- Faster releases

Together: Push code → Tests run → App deploys

The Trust Problem

CI/CD exists because humans are unreliable. We forget tests. We skip review. We say "I'll fix it later."

CI/CD is a robot that never forgets, never gets tired, never skips steps.

```
Without CI/CD:          With CI/CD:
Human: "I ran the       Robot: "Tests failed on line 42."
tests, trust me!"              ↓
       ↓                Production stays safe
Production breaks
```

Git Review: The Basics

Git tracks changes to your code:

```
# Initialize a repository
git init

# Check status
git status

# Stage changes
git add file.py

# Commit changes
git commit -m "Add feature X"

# Push to GitHub
git push origin main
```

Git: Three Areas

Area	Description
Working Directory	Your files on disk
Staging Area	Changes ready to commit
Repository	Committed history

```
# Working → Staging
git add file.py

# Staging → Repository
git commit -m "message"

# Repository → Remote (GitHub)
git push origin main
```

Branching: Work in Isolation

Create a branch for new features:

```
# Create and switch to new branch
git checkout -b feature/add-validation

# Make changes and commit
git add .
git commit -m "Add input validation"

# Push branch
git push origin feature/add-validation
```

Why branches?

- Don't break `main` while experimenting
- Multiple people can work simultaneously
- Easy to undo if something goes wrong

The Parallel Universe Analogy

A branch is like creating a parallel universe. Experiment wildly - main stays safe.

```
main (safe universe)
  │
  ├──→ feature/risky-experiment
  │            │
  │            ├── Try crazy things, break stuff
  │            └── Success! → Merge back to main
  │                                  ↓
  └─────────────────────────────(main improved!)
```

When it fails? Just delete that universe. No harm done.

Pull Requests (PRs)

A PR asks to merge your branch into main:

1. Push your branch to GitHub
2. Click "Create Pull Request"
3. Describe your changes
4. Request a review
5. Address feedback
6. Merge when approved

Why PRs?

- Code review catches bugs
- Discussion before merging
- History of why changes were made

A Simple Workflow

```
main
  │
  ├── Create branch: feature/new-model
  │     │
  │     ├── Write code
  │     ├── Test locally
  │     ├── Push branch
  │     └── Create PR
  │
  ├── Review and discuss
  │
  └── Merge to main
```

This is called "Feature Branch Workflow"

GitHub Actions: Automate Everything

GitHub Actions runs code when events happen:

- On `push` → Run tests
- On `pull_request` → Check code quality
- On `schedule` → Run nightly jobs
- On `release` → Deploy to production

Where? In `.github/workflows/` folder

Your First GitHub Action

Create `.github/workflows/test.yml`:

```yaml
name: Run Tests

on: [push, pull_request]

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

      - name: Set up Python
        uses: actions/setup-python@v5
        with:
          python-version: '3.10'

      - name: Install dependencies
        run: pip install pytest

      - name: Run tests
```

Understanding the Workflow

```yaml
name: Run Tests                     # Name of the workflow

on: [push, pull_request]            # When to run

jobs:
  test:                             # Job name
    runs-on: ubuntu-latest          # Use Ubuntu VM

    steps:
      - uses: actions/checkout@v4   # Get your code

      - name: Set up Python         # Install Python
        uses: actions/setup-python@v5
        with:
          python-version: '3.10'

      - run: pip install pytest     # Install dependencies

      - run: pytest tests/          # Run tests
```

Writing Tests with pytest

Create `tests/test_model.py`:

```python
import pytest
from src.model import predict

def test_predict_returns_valid_output():
    result = predict({"budget": 100, "runtime": 120})
    assert "prediction" in result
    assert result["prediction"] in ["Success", "Risky"]

def test_predict_invalid_input():
    with pytest.raises(ValueError):
        predict({"budget": -100})

def test_predict_confidence_range():
    result = predict({"budget": 100, "runtime": 120})
    assert 0 <= result["confidence"] <= 1
```

Running Tests Locally

Install pytest:

```
pip install pytest
```

Run all tests:

```
pytest tests/ -v
```

Output:

```
tests/test_model.py::test_predict_returns_valid_output PASSED
tests/test_model.py::test_predict_invalid_input PASSED
tests/test_model.py::test_confidence_range PASSED

3 passed in 0.05s
```

CI Workflow for ML Projects

```yaml
name: ML CI Pipeline

on: [push, pull_request]

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

      - uses: actions/setup-python@v5
        with:
          python-version: '3.10'

      - name: Install dependencies
        run: pip install -r requirements.txt

      - name: Run linter (code quality)
        run: ruff check .

      - name: Run tests
        run: pytest tests/ -v
```

Adding Code Quality Checks

Ruff: Fast Python linter

```
pip install ruff
```

Check your code:

```
ruff check .
```

Fix automatically:

```
ruff check --fix .
```

Add to CI workflow:

```
- name: Lint with Ruff
  run: ruff check .
```

Viewing Results on GitHub

After pushing:

1. Go to your repository
2. Click the **Actions** tab
3. See workflow runs

Status indicators:

- ✅ Green check = Tests passed
- ❌ Red X = Tests failed
- 🟡 Yellow = Running

On PRs: GitHub shows status before merging

Secrets: Keeping API Keys Safe

Never commit secrets to Git!

Add secrets in GitHub:

1. Go to Settings → Secrets and variables → Actions
2. Click "New repository secret"
3. Add name and value

Use in workflow:

```
- name: Deploy
  env:
    API_KEY: ${{ secrets.API_KEY }}
  run: python deploy.py
```

Environment Variables

For configuration that varies:

```yaml
jobs:
  test:
    runs-on: ubuntu-latest
    env:
      ENVIRONMENT: testing
      LOG_LEVEL: debug

    steps:
      - name: Run tests
        run: pytest tests/
```

Access in Python:

```python
import os
env = os.getenv("ENVIRONMENT", "development")
```

Caching Dependencies

Speed up builds by caching pip packages:

```
- name: Cache pip packages
  uses: actions/cache@v3
  with:
    path: ~/.cache/pip
    key: ${{ runner.os }}-pip-${{ hashFiles('requirements.txt') }}

- name: Install dependencies
  run: pip install -r requirements.txt
```

Result: First run takes 2 minutes, subsequent runs take 10 seconds

Why Caching is Magic

Every CI run starts with a blank slate. A fresh VM with nothing installed. Without caching, you download the same 500MB of packages on every single push. Caching is like leaving your tools at the job site instead of bringing them home every night. Same result, 10x faster.

Without Caching	With Caching
Download numpy (50MB)	Cache hit! (instant)
Download pandas (30MB)	Cache hit! (instant)
Download sklearn (25MB)	Cache hit! (instant)
Total: 2 minutes	**Total: 10 seconds**

The key: Hash of requirements.txt determines if cache is valid.

Matrix Testing

Test across multiple Python versions:

```yaml
jobs:
  test:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        python-version: ['3.9', '3.10', '3.11']

    steps:
      - uses: actions/setup-python@v5
        with:
          python-version: ${{ matrix.python-version }}

      - run: pytest tests/
```

Creates 3 parallel jobs, one for each Python version

Saving Artifacts

Save files from your workflow:

```
- name: Run tests with coverage
  run: pytest tests/ --cov=src --cov-report=html

- name: Upload coverage report
  uses: actions/upload-artifact@v3
  with:
    name: coverage-report
    path: htmlcov/
```

Download from Actions tab after workflow completes

Example: Complete ML CI/CD

```yaml
name: ML Pipeline

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

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: '3.10'

      - run: pip install -r requirements.txt
      - run: ruff check .
      - run: pytest tests/ -v
```

Example: Deploy on Main Branch

```yaml
deploy:
  needs: test  # Only run if tests pass
  if: github.ref == 'refs/heads/main'  # Only on main branch
  runs-on: ubuntu-latest

  steps:
    - uses: actions/checkout@v4

    - name: Deploy to server
      env:
        DEPLOY_KEY: ${{ secrets.DEPLOY_KEY }}
      run: |
        echo "Deploying to production..."
        # Your deployment script here
```

Pre-commit Hooks

Run checks before you commit:

```
pip install pre-commit
```

Create `.pre-commit-config.yaml`:

```yaml
repos:
  - repo: https://github.com/astral-sh/ruff-pre-commit
    rev: v0.1.0
    hooks:
      - id: ruff
        args: [--fix]
```

Install hooks:

```
pre-commit install
```

Now: Ruff runs automatically on every commit!

Git Best Practices

1. Commit often with clear messages

```
git commit -m "Add input validation for budget field"
```

2. Use branches for features

```
git checkout -b feature/add-auth
```

3. Pull before push

```
git pull origin main
git push origin main
```

4. Never commit secrets

- Use `.gitignore` for `.env` files
- Use GitHub Secrets for API keys

.gitignore for ML Projects

```
# Python
__pycache__/
*.pyc
.venv/

# Data (too large for Git)
data/*.csv
data/*.parquet
*.pkl

# Secrets
.env
secrets.yaml

# IDE
.vscode/
.idea/

# Jupyter
.ipynb_checkpoints/
```

CI/CD Benefits for ML

Without CI/CD	With CI/CD
"Works on my machine"	Works everywhere
Manual testing (or none)	Automated tests
Deploy takes 30 min	Deploy takes 2 min
Break production by accident	Catch bugs before deploy
"Did anyone test this?"	Tests run automatically

Common CI/CD Patterns

1. Test on every push

- Catch bugs immediately
- Fast feedback loop

2. Deploy on merge to main

- Only tested code reaches production
- Automatic deployment

3. Schedule nightly tests

- Check for dependency updates
- Run longer integration tests

4. Manual approval for production

Debugging Failed Workflows

When CI fails:

1. Click on the failed workflow in Actions tab
2. Expand the failed step
3. Read the error message
4. Fix locally and push again

Common issues:

- Missing dependencies in `requirements.txt`
- Tests pass locally but fail in CI (environment differences)
- Secrets not configured

Summary

Concept	Purpose
Git	Track code changes
Branches	Work in isolation
PRs	Review before merging
GitHub Actions	Automate workflows
pytest	Write and run tests
Secrets	Store sensitive data
Caching	Speed up CI

Lab Preview

This week you'll:

1. Set up a Git repository with proper structure
2. Write tests for your ML code
3. Create a GitHub Actions workflow
4. Add code quality checks (Ruff)
5. Configure caching for faster builds
6. Set up pre-commit hooks

Result: Automated testing on every push!

Questions?

Key takeaways:

- CI/CD automates testing and deployment
- GitHub Actions is free for public repos
- Always test before deploying
- Secrets should never be in code

Next week: Edge Deployment