

Git, GitHub Actions & CI/CD

CS 203: Software Tools and Techniques for AI

Prof. Nipun Batra, IIT Gandhinagar

Why Automate?

Manual Process:

1. Write code
2. Run tests locally (maybe?)
3. Commit
4. SSH into server
5. `git pull`
6. Restart service

Automated (CI/CD):

1. `git push`
2. **CI**: Tests run, linters check code.
3. **CD**: If tests pass, auto-deploy to server.

The DevOps Culture

Traditional: Development throws code over the wall to Operations

DevOps: Developers own deployment and monitoring

Key principles:

1. **Automation:** Manual work is error-prone
2. **Version everything:** Code, config, infrastructure
3. **Continuous feedback:** Fast iteration cycles
4. **Collaboration:** Break down silos
5. **Monitoring:** Know when things break

MLOps = DevOps + Data + Models

CI vs CD vs CD

Continuous Integration (CI):

- Automatically merge code frequently (daily)
- Run automated tests on every commit
- **Goal:** Catch bugs early

Continuous Delivery:

- Code is always deployable
- Manual approval needed for production
- **Goal:** Reduce deployment risk

Continuous Deployment:

- Every change automatically goes to production
- Zero human intervention

Git Fundamentals Review

Three areas:

1. **Working Directory:** Your local files
2. **Staging Area:** Changes ready to commit
3. **Repository:** Committed history

Basic workflow:

```
git add file.py           # Working → Staging
git commit -m "msg"       # Staging → Repository
git push origin main      # Repository → Remote
```

Why it matters for CI/CD: Git is the trigger for automation.

Branching Strategies

Feature Branch Workflow:

```
git checkout -b feature/add-auth  
# Make changes  
git commit -m "Add authentication"  
git push origin feature/add-auth  
# Create PR → Review → Merge
```

Benefits:

- Isolate features
- Easy code review
- Revert if needed

Branch protection: Require PR reviews, passing tests before merge.

Gitflow vs Trunk-Based

Aspect	Gitflow	Trunk-Based
Branches	main, develop, feature/*	main only
Release	release/* branches	Tags on main
Complexity	High	Low
Best for	Scheduled releases	Continuous deployment

Modern trend: Trunk-based with feature flags.

Pull Request Best Practices

Good PR:

1. **Small:** < 400 lines changed
2. **Focused:** One feature/bug fix
3. **Tested:** All tests pass
4. **Documented:** Clear description

Review checklist:

- Code style consistent?
- Tests added/updated?
- No hardcoded secrets?
- Performance implications?

Automate: Use bots for formatting, test coverage checks.

GitHub Automation Ecosystem

1. **Git:** Version control (branches, merges).
2. **GitHub API:** Programmatic access to repos (Issues, PRs).
3. **GitHub Actions:** Serverless compute to run workflows.
4. **Webhooks:** Event-driven triggers.
5. **GitHub Apps:** Reusable automation packages
6. **GitHub Copilot:** AI pair programming

Part 1: GitHub API & PyGithub

Automating the "boring" stuff.

Authentication:

- Use **Personal Access Tokens (PATs)** (Fine-grained).
- Store in `.env` (Never commit!).

```
from github import Github
import os

g = Github(os.getenv("GITHUB_TOKEN"))
repo = g.get_repo("nipunbatra/stt-ai-teaching")

# List open issues
for issue in repo.get_issues(state='open'):
    print(issue.title)
```

Use Case: AI Code Reviewer

Idea: When a PR is opened, fetch the diff, send to LLM, post comment.

```
pr = repo.get_pull(123)
files = pr.get_files()

for file in files:
    # Get the changes
    patch = file.patch

    # Analyze with LLM
    review = llm.review_code(patch)

    # Post comment
    pr.create_issue_comment(f"## AI Review for {file.filename}\n\n{review}")
```

Part 2: GitHub Actions (CI)

Workflows: YAML files in `.github/workflows/`.

Structure:

- **Triggers:** `on: push`, `on: pull_request`
- **Jobs:** Parallel tasks (`test`, `lint`, `deploy`)
- **Steps:** Sequential commands within a job.

Example: CI Pipeline for Python

```
name: CI

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 ruff

      - name: Lint with Ruff
        run: ruff check .

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

GitHub Actions: Triggers

Event types:

```
on:
  push:
    branches: [main, develop]
  pull_request:
    branches: [main]
  schedule:
    - cron: '0 0 * * *' # Daily at midnight
  workflow_dispatch:    # Manual trigger
  release:
    types: [published]
```

Conditional execution:

```
jobs:
  deploy:
    if: github.ref == 'refs/heads/main'
    # Only runs on main branch
```

Caching Dependencies

Speed up builds by caching:

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

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

Typical speedup: 2-5 minutes → 30 seconds.

Artifacts and Outputs

Save build artifacts:

- name: Upload test results
uses: actions/upload-artifact@v3
with:
 name: pytest-results
 path: test-results/
- name: Upload model
uses: actions/upload-artifact@v3
with:
 name: trained-model
 path: model.pkl

Download in later jobs or from GitHub UI.

CI/CD for ML

ML is harder than standard software.

- **Data Tests:** Validate schema (Pydantic/Great Expectations).
- **Model Tests:** Check performance threshold.
- **Hardware:** Need GPU runners? (Self-hosted runners).

Matrix Testing:

Test across python versions / OSs.

```
strategy:
  matrix:
    python-version: ["3.9", "3.10", "3.11"]
    os: [ubuntu-latest, windows-latest]
```

ML-Specific CI Checks

Data validation:

- name: Validate data schema
run: |
python -m pytest tests/test_data_schema.py
- name: Check for data drift
run: |
python scripts/detect_drift.py

Model performance gates:

- name: Train and evaluate
run: |
python train.py
python evaluate.py
- name: Check accuracy threshold
run: |
accuracy=\$(cat metrics.json | jq '.accuracy')

Docker Integration

Build and push Docker images:

```
- name: Build Docker image
  run: docker build -t myapp:${{ github.sha }} .

- name: Push to registry
  run: |
    echo ${ secrets.DOCKER_PASSWORD } | docker login -u ${ secrets.DOCKER_USERNAME } --password-stdin
    docker push myapp:${{ github.sha }}
```

Benefits:

- Consistent environment
- Easy deployment
- Reproducible builds

Deployment Strategies

Blue-Green Deployment:

- Two identical environments (blue, green)
- Deploy to inactive, then switch traffic
- **Rollback:** Just switch back

Canary Deployment:

- Deploy to small % of users first
- Monitor metrics
- Gradually increase if healthy

Rolling Deployment:

- Update instances one by one
- Always some instances serving traffic

GitHub Environments

Separate configs for dev/staging/prod:

```
jobs:
  deploy:
    environment:
      name: production
      url: https://myapp.com
    steps:
      - name: Deploy
        run: ./deploy.sh
```

Environment protection rules:

- Require reviewers
- Wait timer before deployment
- Deployment branches (only main)

Monitoring CI/CD Pipelines

Key metrics:

1. **Build time:** How long does CI take?
2. **Success rate:** % of builds passing
3. **Mean time to recovery:** How fast do you fix broken builds?
4. **Deployment frequency:** How often do you deploy?

Tools:

- GitHub Actions insights
- Custom metrics (Prometheus + Grafana)
- Alerts (Slack, email)

Secrets Management

Never hardcode API keys.

1. Add secret in GitHub Repo Settings -> Secrets.
2. Access in Action:

```
env:  
  OPENAI_API_KEY: ${ secrets.OPENAI_API_KEY }
```

Part 3: Testing Strategy

The Testing Pyramid for AI:

1. **Unit Tests (Most):** Test individual functions (`preprocess_text()`).
2. **Integration Tests:** Test model loading + inference flow.
3. **System Tests:** API endpoints (FastAPI `TestClient`).
4. **Data Tests:** Validate input distributions.

Writing Tests with Pytest

```
# test_model.py
from my_model import predict

def test_prediction_shape():
    output = predict([1.0, 2.0])
    assert "class" in output
    assert output["confidence"] >= 0.0

def test_model_invariance():
    # Adding noise shouldn't flip prediction
    out1 = predict(x)
    out2 = predict(x + epsilon)
    assert out1["class"] == out2["class"]
```

Pre-commit Hooks

Run checks *before* you push.

Prevents bad code from even reaching the repo.

`.pre-commit-config.yaml` :

```
repos:
-   repo: https://github.com/psf/black
    rev: 23.3.0
    hooks:
    -   id: black
```

Workflow: `git commit` -> Black runs -> Formatting fixed -> Commit succeeds.

Git Branching Strategies

Trunk-Based Development: Everyone commits to main frequently.

- **Pros:** Simple, fast feedback
- **Cons:** Requires discipline, good tests
- **Used by:** Google, Facebook

GitFlow: Feature branches → develop → release → main.

```
main (production)
  ↑
release/1.2
  ↑
develop
  ↑
feature/new-model
```

GitHub Flow (simpler):

Advanced Git: Rebasing vs Merging

Merge: Creates merge commit.

```
git checkout main
git merge feature-branch
# Creates merge commit with two parents
```

Rebase: Replay commits on top of main.

```
git checkout feature-branch
git rebase main
# Re-writes history, linear timeline
```

When to use:

- **Merge:** Public branches, preserves history
- **Rebase:** Local branches, clean history

Golden rule: Never rebase public branches!

Infrastructure as Code (IaC)

Problem: Manual server setup is error-prone.

Solution: Define infrastructure in code.

Terraform example:

```
resource "aws_instance" "ml_server" {
  ami           = "ami-0c55b159cbfafa1f0"
  instance_type = "p3.2xlarge" # GPU instance

  tags = {
    Name = "ML-Training-Server"
  }
}

resource "aws_s3_bucket" "model_artifacts" {
  bucket = "my-ml-models"
  versioning {
    enabled = true
  }
}
```

Blue-Green Deployment

Strategy: Run two identical environments.

```
Blue (current production) ← 100% traffic
Green (new version)       ← 0% traffic
```

```
# After testing:
```

```
Blue ← 0% traffic
```

```
Green ← 100% traffic # Switch!
```

```
# If issues, instant rollback:
```

```
Blue ← 100% traffic # Switch back!
```

Implementation (with load balancer):

```
# GitHub Actions
```

```
- name: Deploy to Green
```

```
  run: |
```

```
    kubectl set image deployment/app app=myapp:${{ github.sha }} -n green
```

```
- name: Wait for health
```

Canary Releases

Strategy: Gradually shift traffic to new version.

```
v1.0 ← 100% traffic  
v2.0 ← 0% traffic  
  
# Step 1:  
v1.0 ← 95% traffic  
v2.0 ← 5% traffic # Test on small subset  
  
# Step 2:  
v1.0 ← 50% traffic  
v2.0 ← 50% traffic  
  
# Step 3 (if metrics good):  
v1.0 ← 0% traffic  
v2.0 ← 100% traffic
```

Monitoring: Watch for errors, latency spikes.

Rollback if metrics degrade:

Feature Flags and Gradual Rollouts

Feature flags: Toggle features without deploying.

```
import flagsmith

flags = flagsmith.get_environment_flags()

if flags.is_feature_enabled("new_model_v2"):
    model = load_model("v2")
else:
    model = load_model("v1")
```

Gradual rollout:

```
# Enable for 10% of users
user_id_hash = hash(user_id) % 100
if user_id_hash < 10:
    use_new_feature = True
```

Benefits:

Rollback Strategies

1. Git revert:

```
git revert HEAD      # Create new commit that undoes last  
git push  
# CI/CD deploys the revert
```

2. Kubernetes rollout:

```
kubectl rollout undo deployment/app  
# Instantly roll back to previous version
```

3. Database rollbacks (harder):

- Forward-compatible migrations
- Blue-green for schema changes
- Keep old code compatible with new schema

CI/CD Security Best Practices

Secrets management:

```
# GitHub Secrets (encrypted)
steps:
  - name: Deploy
    env:
      AWS_ACCESS_KEY: ${ secrets.AWS_ACCESS_KEY }
      DB_PASSWORD: ${ secrets.DB_PASSWORD }
```

Never commit secrets!

SAST (Static Application Security Testing):

```
- name: Run Bandit (Python security linter)
  run: bandit -r src/

- name: Check for secrets
  uses: trufflesecurity/trufflehog@main
```

Matrix Builds for Multi-Environment Testing

Test across multiple configurations:

```
strategy:
  matrix:
    python-version: [3.8, 3.9, 3.10, 3.11]
    os: [ubuntu-latest, macos-latest, windows-latest]

steps:
  - uses: actions/setup-python@v4
    with:
      python-version: ${ matrix.python-version }

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

Result: 4 Python versions × 3 OS = 12 test jobs (parallel).

ML-specific matrix:

```
matrix:
```

Caching Strategies in CI/CD

Problem: Installing dependencies is slow (2-5 minutes).

Solution: Cache dependencies.

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

- name: Install dependencies
  run: pip install -r requirements.txt
# Cached, takes 10 seconds instead of 2 minutes
```

Docker layer caching:

```
- name: Build Docker image
  uses: docker/build-push-action@v4
  with:
    cache-from: type=gha
```

Summary

1. **GitHub API:** Automate repo management and reviews.
2. **GitHub Actions:** The engine for CI/CD.
3. **Testing:** Essential for robust ML systems (Unit, Data, Model tests).
4. **Pre-commit:** The first line of defense.

Advanced Topics:

- Git branching strategies (trunk-based, GitFlow)
- Advanced Git (rebase, interactive rebase)
- Infrastructure as Code (Terraform)
- Deployment strategies (blue-green, canary)
- Feature flags and gradual rollouts
- Rollback strategies