Continuous Integration and Continuous Delivery (CI/CD)

Automating Software Delivery

Mehdi Lotfinejad

🎯 Learning Objectives

By the end of this lesson, you will be able to:

1. **Define** Continuous Integration (CI) and Continuous Delivery (CD) and explain their core purposes
2. **Identify** the key phases in a CI/CD pipeline
3. **Explain** how automation improves software quality and delivery speed
4. **Recognize** the difference between CI and CD in practice
5. **Set up** a basic CI/CD workflow using GitHub Actions

The Problem: Integration Hell 🔥

Traditional Software Development:

- Developers work independently for weeks
- Code integration happens at the end
- Conflicts emerge everywhere
- Bugs multiply exponentially
- Integration takes days or weeks

> **Result:** Expensive disasters, missed deadlines, and stressed teams

The Modern Solution: CI/CD

Continuous Integration and Continuous Delivery (CI/CD) automates the process of integrating code changes, testing them, and delivering software to users.

Think of it as daily inspections:

- Problems caught immediately when they're easy to fix
- Not weeks later when they're expensive disasters
- Teams merge code changes multiple times daily
- Automated testing catches bugs before users see them

Why CI/CD Matters to You

Real-World Impact:

- **Bug fixes in minutes:** Critical issues resolved and deployed within hours
- **Faster releases:** New features reach users quickly
- **Higher quality:** Automated testing catches bugs early
- **Less stress:** No more 2 AM manual deployments

> **Industry Standard:** Every modern development team uses CI/CD. It's a fundamental skill employers expect!

Real-World Examples

Companies using CI/CD:

- **Instagram:** Multiple deployments per day
- **Netflix:** Thousands of production deployments daily
- **Gmail:** Continuous feature updates
- **Amazon:** Deployment every 11.7 seconds (at peak)

Without CI/CD, this would be impossible!

Understanding Continuous Integration (CI)

Continuous Integration is the practice of automatically integrating code changes from multiple developers into a shared repository several times per day.

Key Points:

- Each integration triggers automated build and test
- Detects problems early
- Keeps code always in a working state

CI: The Restaurant Kitchen Analogy

Without CI (Old Way):

- Each chef works independently
- Nothing is tasted until done
- Food is cold/burnt at serving
- Head chef fixes for hours
- Hungry customers wait

With CI (Modern Way):

- Chefs work step-by-step
- Quality checked after each step
- Problems caught immediately
- Everything perfect at serving
- Happy customers!

Problems CI Solves (Part 1)

Code Conflicts:

- Two developers modify the same file
- Changes are incompatible
- Hours spent resolving conflicts

Hidden Bugs:

- Code works alone
- Breaks when combined
- Hard to track down the source

Problems CI Solves (Part 2)

Integration Delays:

- Days/weeks resolving conflicts
- Time not spent building features
- Project timelines slip

Finger-Pointing:

- Hard to identify who caused problems
- Team morale suffers
- Blame culture develops

> **CI Solution:** Integrate frequently, test automatically, catch problems early!

The CI Pipeline: Overview

Think of it as a factory assembly line:

Each station has a specific job, and the product only moves forward if it passes quality checks.

Five Phases:

1. Git Clone
2. Build/Compile
3. Unit Test
4. Package
5. Report

Phase 1: Git Clone - Getting the Code

What happens:

- Developer pushes code to GitHub
- CI system detects the change
- Latest code is downloaded (cloned)

Analogy: Factory receiving raw materials before building anything

```
name: CI Pipeline
on:
  push:
    branches: [ main, develop ]
  pull_request:
    branches: [ main ]
```

Phase 2: Build/Compile - Putting It Together

What happens:

- CI system compiles code (if needed)
- Installs all dependencies
- Checks for basic errors

Analogy: Assembling IKEA furniture—make sure all pieces are present

```
# Install dependencies
pip install -r requirements.txt

# Check for syntax errors
python -m py_compile src/*.py
```

Phase 2: What If Build Fails?

Pipeline stops immediately!

Common causes:

- Typo in your code
- Missing dependency
- Incompatible library versions

> **Remember:** Catching errors here is much cheaper than catching them in production!

Phase 3: Automated Testing - Quality Control

What happens:

- System runs automated tests
- Verifies code works correctly
- Tests are like quality inspectors

Analogy: Before shipping a phone, manufacturers test every button, camera, battery life, etc.

Types of tests:

- **Unit tests:** Test individual functions
- **Integration tests:** Test how parts work together
- **Code quality checks:** Style guidelines and best practices

Phase 3: Example Test

```python
def add_numbers(a, b):
    """Add two numbers and return the result."""
    return a + b

def test_add_numbers():
    """Test the add_numbers function."""
    assert add_numbers(2, 3) == 5      # Positive
    assert add_numbers(-1, 1) == 0     # Negative
    assert add_numbers(0, 0) == 0      # Edge case
```

Tests are not optional—they're essential!

Phase 4: Package - Preparing for Delivery

What happens if tests pass:

The system packages the application into deployable format:

- Docker container (self-contained package)
- ZIP file with compiled code
- Executable file

Analogy: Putting your product in a shipping box with instructions, ready to send to customers

Phase 5: Report - Feedback Loop

Results sent to the team:

- ✅ **Green (Success):** All tests passed!
- ❌ **Red (Failure):** Something broke, here's what and where
- ⚠️ **Yellow (Warning):** Tests passed but concerns exist

Where you see feedback:

- Email notifications
- Slack messages
- GitHub pull request checks
- CI dashboard

Why Automation Matters

Manual processes are:

- Slow and time-consuming
- Error-prone (humans make mistakes)
- Inconsistent (different each time)
- Don't scale with team growth

Automated processes:

- Execute same steps perfectly every time
- Eliminate human error
- Provide consistent, repeatable results
- Scale effortlessly

Understanding Continuous Delivery (CD)

Continuous Delivery extends CI by automatically preparing code for release to production.

What it does:

- Every change that passes CI is deployed to staging
- Testing in production-like conditions
- Code is always ready to ship to customers

CI vs. CD: The Key Difference

Continuous Integration (CI):

- Focuses on integration and testing
- Ensures code works correctly
- Answers: **"Does it work?"**

Continuous Delivery (CD):

- Focuses on deployment readiness
- Ensures code can be released anytime
- Answers: **"Is it ready to ship?"**

CD Pipeline Extension

CD adds three stages after CI:

1. **Staging Deployment**
    - Code deployed to staging environment
    - Copy of production for testing
    - No impact on real users
2. **Integration Testing**
    - Test with real databases, APIs, services
    - Verify production-like behavior
3. **Production Deployment**
    - Deploy to servers where real users access app

Continuous Delivery vs. Continuous Deployment

Continuous Delivery:

- Code is always ready to deploy
- Humans decide when to release
- Manual approval for production
- **Most common approach**

Continuous Deployment:

- Every change automatically goes to production
- No human intervention
- Requires high confidence in tests
- **Advanced teams only**

Environment Strategy: Three-Stage Safety Net

```
Development → Staging → Production
    ↓             ↓          ↓
  Developers    Testing    Real Users
```

Movie Production Analogy:

1. **Development:** Filming scenes
2. **Staging:** Screen test with focus groups
3. **Production:** Movie releases in theaters

Why Multiple Environments Matter

Development:

- Fast, isolated workspace
- Break things safely
- Rapid experimentation

Staging:

- Realistic testing environment
- Production-like data and configurations
- Final validation before release

Production:

- Real users, real data, real consequences
- No room for errors
- Full monitoring and alerts

Real CI/CD Example: Flask API

```python
# app.py
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/hello/<name>')
def hello(name):
    """Return a personalized greeting."""
    return jsonify({'message': f'Hello, {name}!'})

@app.route('/health')
def health():
    """Health check endpoint."""
    return jsonify({'status': 'healthy'})
```

Example: Automated Tests

```python
# test_app.py
import pytest
from app import app

@pytest.fixture
def client():
    """Create a test client for the Flask app."""
    app.config['TESTING'] = True
    with app.test_client() as client:
        yield client

def test_hello_endpoint(client):
    """Test the hello endpoint."""
    response = client.get('/hello/World')
    assert response.status_code == 200
    assert response.json['message'] == 'Hello, World!'
```

GitHub Actions Workflow (Part 1)

```yaml
name: CI/CD Pipeline

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

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v3
```

GitHub Actions Workflow (Part 2)

```
      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.9'

      - name: Install dependencies
        run: |
          pip install flask pytest

      - name: Run tests
        run: |
          pytest test_app.py -v
```

GitHub Actions Workflow (Part 3)

```
  deploy_staging:
    needs: test  # Only runs if tests pass
    if: github.ref == 'refs/heads/main'
    runs-on: ubuntu-latest
    steps:
      - name: Deploy to staging
        run: |
          echo "Deploying to staging..."
          # Actual deployment commands here
```

How the Pipeline Works: Timeline

Seconds 0-5: Trigger

- Developer pushes code
- GitHub detects the push
- Pipeline starts automatically

Seconds 5-60: Test Job

- ✓ Checkout code [5s]
- ✓ Set up Python [10s]
- ✓ Install dependencies [20s]
- ✓ Run tests [15s]

Pipeline Timeline Continued

Second 60: Decision Point

- **If tests fail:** Stop here, notify developer
- **If tests pass:** Continue to deployment

Seconds 60-120: Deploy Job (main branch only)

- ✓ Deploy to staging [60s]
- ✓ Health check [10s]
- ✓ Notify team [2s]

Second 120: Feedback

- Green checkmark ✓ on GitHub
- Team notifications sent

Understanding the Configuration: Triggers

```
on:
  push:
    branches: [ main ]      # Run on pushes to main
  pull_request:
    branches: [ main ]      # Run on PRs to main
```

Meaning:

"Run tests on every push to main AND every pull request targeting main."

Understanding the Configuration: Jobs

```
jobs:
  test:                       # First job: testing
    runs-on: ubuntu-latest  # Use Ubuntu Linux

  deploy_staging:
    needs: test               # Wait for test to succeed
    if: github.ref == 'refs/heads/main'  # Only main
```

The `needs: test` creates a dependency:

Deployment can't happen until tests pass!

Why This Pipeline Works

- ✅ **Automatic:** Runs without human intervention
- ✅ **Fast:** Completes in under 2 minutes
- ✅ **Comprehensive:** Tests everything before deployment
- ✅ **Safe:** Deploys only after tests pass
- ✅ **Transparent:** Clear feedback on success/failure
- ✅ **Auditable:** History of all runs preserved

Common Pitfall #1: Skipping Tests

The Mistake:

Setting up CI/CD without comprehensive tests

Why It's Dangerous:

- CI/CD only as good as your tests
- Without tests, you automate deploying bugs
- Users find problems instead of tests

> **Real Example:** Company deployed broken code multiple times per day. Users were furious!

Pitfall #1: The Solution

Follow the testing pyramid:

- Many unit tests (fast and easy)
- Some integration tests (moderate)
- Few end-to-end tests (slow but comprehensive)

Aim for 80% code coverage

> **Remember:** Slow down to speed up. Time spent writing tests saves hours of debugging!

Common Pitfall #2: Ignoring Failed Builds

The Mistake:

Seeing red "failed" status and thinking "I'll fix it later"

Why It's Dangerous:

- Broken pipeline = broken smoke detector
- New bugs accumulate undetected
- Can't identify which change caused problems

Analogy: Driving with "check engine" light on—eventually the car breaks down completely!

Pitfall #2: The Solution

Treat failed builds as P0 emergencies:

1. **Stop:** Don't add new features
2. **Investigate:** Read error messages
3. **Fix:** Correct the problem
4. **Verify:** Ensure pipeline goes green
5. **Learn:** Prevent future occurrences

> **Team Rule:** "Don't break the build, and if it breaks, fix it immediately!"

Common Pitfall #3: Over-Complicated Setup

The Mistake:

Implementing all CI/CD best practices immediately

Why It's Dangerous:

- Creates maintenance burden
- Debugging becomes impossible
- Team can't understand the pipeline
- More time managing than building features

Pitfall #3: The Solution - Phased Approach

Phase 1 (Week 1): Basic CI only

- Run tests on every push
- Get familiar with feedback loop

Phase 2 (Weeks 2-3): Add staging deployment

- Automatic deployment to test environment
- Manual approval for production

Phase 3 (Month 2+): Advanced features

- Code quality checks
- Performance testing
- Automatic production deployment

Common Pitfall #4: Hardcoding Secrets

The Mistake:

Putting passwords, API keys in code/config

Why It's Dangerous:

- Secrets become public
- Anyone with repo access can steal credentials
- Data breaches and security incidents

```
# ❌ BAD - Never do this!
deploy:
  - aws deploy --key AKIAIOSFODNN7EXAMPLE
```

Pitfall #4: The Solution

Use secret management:

- GitHub Secrets
- AWS Secrets Manager
- HashiCorp Vault
- Azure Key Vault

Best practices:

- Never commit credentials to code
- Rotate secrets regularly
- Use environment variables
- Implement access controls

Common Pitfall #5: No Rollback Strategy

The Mistake:

Focusing only on deploying forward

Why It's Dangerous:

- Eventually, a deployment will break production
- Without rollback plan, you'll panic
- Situation gets worse, not better

Pitfall #5: The Solution

Implement rollback procedures:

- Tag releases with version numbers
- Keep previous versions readily deployable
- Practice rolling back in staging
- Document rollback procedure
- Consider blue-green deployments

> **Remember:** Hope for the best, plan for the worst!

Key Takeaways: Core Concepts

- **CI/CD automates** integration, testing, and deployment
- **CI asks:** "Does this code work?"
- **CD asks:** "Is this code ready to ship?"
- Pipeline phases: Git Clone → Build → Test → Package → Report

Key Takeaways: Best Practices

Start simple:

- Basic CI first (build and test)
- Gradually add CD capabilities

Good tests are essential:

- CI/CD is only as good as your tests
- Write tests before complex automation

Fix broken builds immediately:

- Don't let problems accumulate
- Keep your safety net working

Key Takeaways: Security & Reliability

Never hardcode secrets:

- Use proper secret management
- Rotate credentials regularly

Have a rollback strategy:

- Deployments will fail
- Be prepared to revert

The Real Value:

- Speed, quality, confidence, peace of mind

The CI/CD Quote

> *"CI/CD is not about tools—it's about culture. It's about teams working together, integrating frequently, testing thoroughly, and delivering value to users continuously."*

Start small, learn from failures, improve incrementally.

Practice Questions

Question 1:

What is the main difference between Continuous Integration and Continuous Delivery?

Question 2:

Name the five typical phases of a CI pipeline in order.

Question 3:

Why is automated testing considered crucial in CI/CD?

Answers (Part 1)

Question 1:

- **CI:** Automatically integrates and tests code changes
- **CD:** Automatically prepares code for deployment
- CI asks "Does it work?", CD asks "Is it ready to ship?"

Question 2:

1. Git Clone
2. Compile/Build
3. Unit Test
4. Package
5. Report

Answers (Part 2)

Question 3:

Automated testing is crucial because:

- **Safety net:** Catches bugs before users
- **Confidence:** Developers can make changes safely
- **Speed:** Fast feedback loop (seconds, not days)
- **Documentation:** Tests show how code should work

> **Without tests, CI/CD just automates deploying broken code faster!**

Security Considerations

Never do:

- ❌ Hardcode passwords/API keys in code
- ❌ Commit secrets to version control
- ❌ Use same credentials across environments
- ❌ Grant unnecessary permissions

Always do:

- ✅ Use secret management systems
- ✅ Implement access controls
- ✅ Use secure protocols (SSH, HTTPS)
- ✅ Rotate secrets regularly
- ✅ Audit access logs

Your Next Steps: Hands-On Learning

🚀 **Ready to build your own CI/CD pipeline?**

We've prepared a complete hands-on tutorial where you'll:

- Build a real CI/CD pipeline from scratch
- Work with GitHub Actions
- Deploy to staging and production
- Implement security best practices

Hands-On Project Details

What you'll build:

1. Fork repository and set up project
2. Write and run automated tests
3. Create GitHub Actions workflow
4. Containerize with Docker
5. Deploy to multiple environments
6. Implement security checks
7. Add monitoring
8. Practice rollback procedures

Prerequisites for Hands-On

Before starting, ensure you have:

- GitHub account (free tier is fine)
- Basic understanding of Git
- Python 3.8 or higher installed
- Text editor (VS Code recommended)

Time commitment:

- 2-3 hours total
- 30 min: Setup and understanding
- 1 hour: Building pipeline
- 1 hour: Testing and experimenting

Learning Approach Tips

Tips for success:

1. Read all instructions carefully
2. Experiment! Try changing things
3. Break things intentionally (learn from errors)
4. Ask questions in repository discussions
5. Share your success when complete

Don't rush—understand WHY each step matters!

Access the Hands-On Tutorial

👉 github.com/datatweets/cicd-pipeline-demo

By the end, you'll have a working CI/CD pipeline that you built yourself!

Final Motivation

Remember: Every expert was once a beginner who decided to keep learning.

Your journey into modern software development practices starts now! 🎯

The best CI/CD pipeline is the one you'll actually use.
Start small, learn from failures, improve incrementally.

Resources and References

Official Documentation:

- Red Hat - What is CI/CD?
- CloudBees - Beginner's Guide to CI/CD
- Atlassian - Continuous Delivery Guide

Learning Resources:

- freeCodeCamp - CI/CD Handbook
- GitHub Actions Documentation
- Docker Documentation

Community:

- GitHub Discussions
- Stack Overflow
- DevOps subreddits

Questions?

Thank you for your attention!

🌐 **lotfinejad.com**

📧 **Contact for questions and discussions**

Next Lesson: Practical CI/CD Implementation Workshop