Columbia Business School

Columbia University
Justice Through Code

# W5D1: CI/CD Fundamentals

# GitHub Actions, Automated Testing, Deployment

Cohort: AISE 2025

Week: 5

Day: 1

Session Code: W5D1

Duration: 3 hours

Phase: Pattern Recognition & Debugging

# Tonight's Concepts

1

## Continuous Integration Basics

**Understanding automated testing pipelines, GitHub Actions workflows, and why CI/CD prevents production disasters**

2

## GitHub Actions Workflows

**Writing YAML workflows that trigger on push/pull requests, understanding jobs, steps, runners, and workflow syntax**

3

## Automated Testing in CI

**Running pytest in cloud environments, observing failing vs. passing tests, and reading CI logs effectively**

4

## Gated Deployments

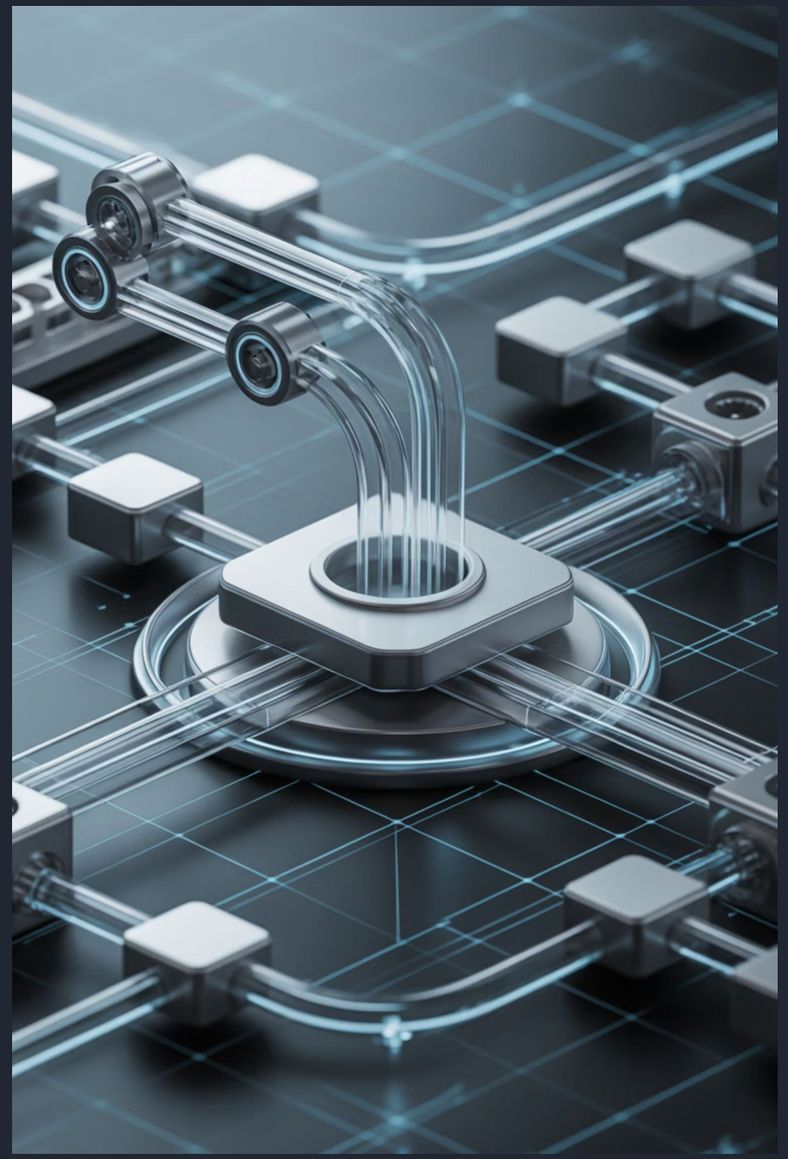
**Using the needs: keyword to create job dependencies, ensuring tests pass before deployment runs**

5

## Matrix Testing & Branch Protection

**Testing across multiple Python versions in parallel, configuring branch protection rules to enforce CI checks**

# The Friday 5 PM Nightmare



## Imagine this scenario...

It's Friday at 5 PM. You've been working on a feature all week. You merge your branch to main, don't run tests, and head home.
Ten minutes later, your phone explodes with notifications.

**Production is down. Users can't log in. The on-call engineer is paging everyone.**

Instead of enjoying your weekend, you're scrambling to roll back changes and debug late into the night.

What would YOU do in this situation?

# What Fellows Think

### Rollback Changes

Revert to the last working version and investigate what went wrong

### Debug All Night

Stay up late trying to fix the issue before Monday morning

### Delay to Monday

Leave it broken over the weekend and deal with it next week

### Hope No One Notices

Cross your fingers and pray the issue resolves itself

> **All of these are REACTIVE.** What if we could prevent this from happening in the first place?

# The Real Solution

### Continuous Integration (CI)

Automatically tests every commit to catch bugs before they reach production

### Continuous Delivery (CD)

Ensures your code is always in a deployable state, ready to ship at any moment

### Continuous Deployment

Automatically ships code to production when all tests pass—no manual intervention

In our Friday scenario, the merge would have been **blocked automatically** because tests would have failed. CI/CD acts as your safety net, preventing disasters before they happen.

Prevention > Reaction

# Why Automate?

### Humans Are Unreliable

We forget steps, skip tests when rushed, and make mistakes under pressure. "Works on my machine" becomes a running joke when manual processes fail.

### Faster Feedback

Get results in minutes, not days. Know immediately if your code breaks something, allowing you to fix issues while the context is fresh in your mind.

### Team Collaboration

Everyone follows the same process. No more "it worked for me" debates. Consistent environments mean consistent results across the entire team.

### Safe Small Releases

Ship confidently and frequently. Small, tested changes are easier to debug and roll back than massive releases that happen once a month.

> Automation = Speed + Safety

# Manual vs Automated Pipeline

## ❌ Manual Process

1. **Write Code**
   Developer completes feature
2. **Manual Testing**
   Run tests locally (maybe)
3. **Manual Review**
   Hope someone checks it
4. **HOPE NOTHING BREAKS**
   Deploy and pray
5. **Production**
   Hours to days later

## ✅ Automated CI/CD

1. **Push Code**
   Developer commits changes
2. **CI Tests Run**
   Automatic validation
3. **Build & Stage**
   Blocks merge on fail
4. **Auto Deploy**
   Only if tests pass
5. **Production**
   2-10 minutes total

The automated pipeline **physically prevents** broken code from reaching production. If tests fail, the merge is blocked—no exceptions.

# Interactive Reflection

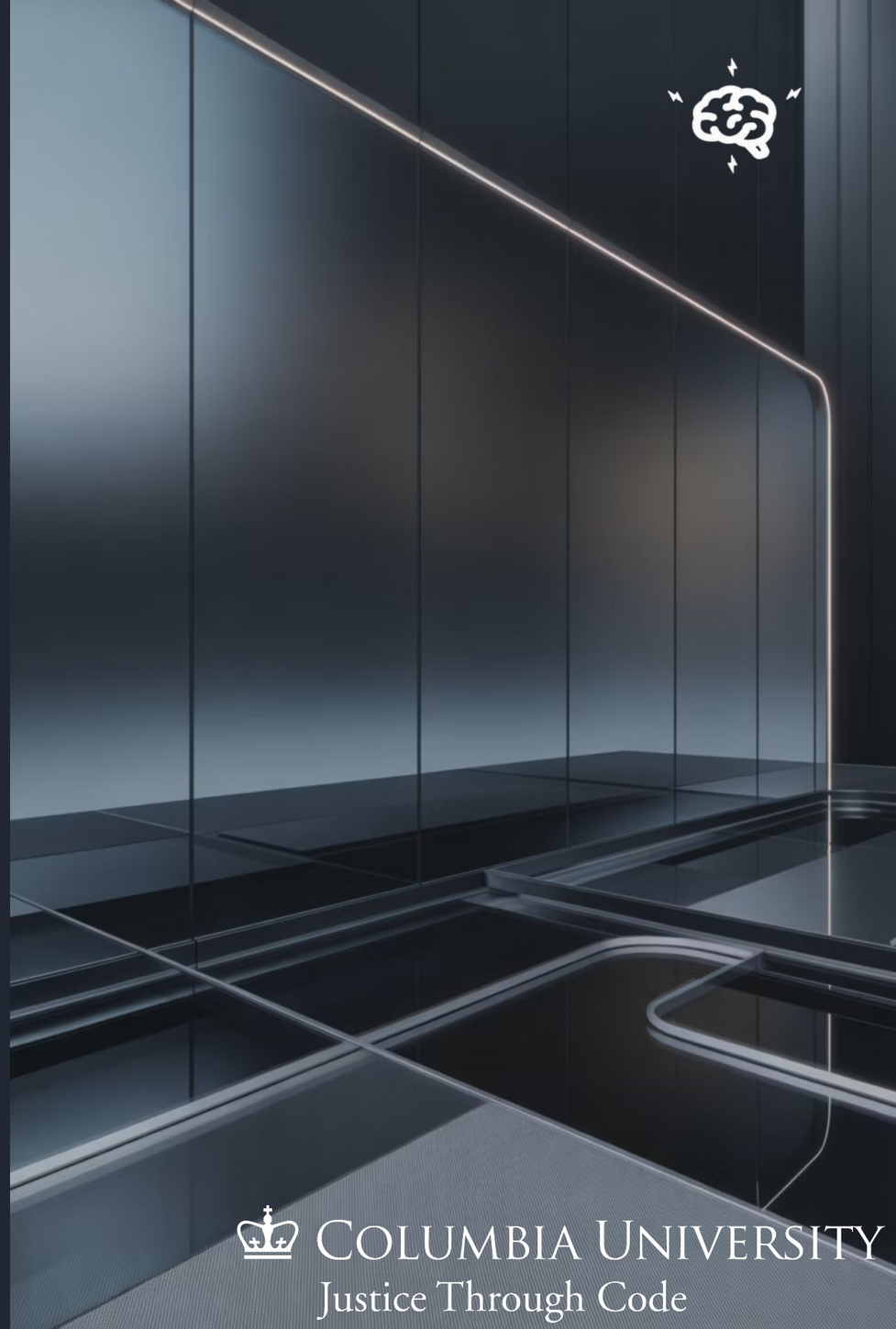
## Question 1

Which step in your current workflow do you most wish you could automate? Think about what takes the most time or causes the most frustration.

## Question 2

What's the worst thing that could happen if you skip running tests before merging? Share a real experience or imagine a nightmare scenario.

**Type your answers in chat!** We'll discuss the most common responses and how CI/CD addresses these pain points.



Key Takeaway

# CI/CD = Confidence + Speed

### Confident Merges

Broken code physically cannot reach main branch

### Consistent Environments

Same tests run the same way, every time

### Faster Releases

Ship in minutes, not hours or days

### Happier Developers

No more Friday night production fires

**Tonight, YOU will build this system.** By the end of class, you'll have a working CI/CD pipeline protecting your code.

# Setup Overview

01

### Create GitHub Repository

Set up a new repo to host our pipeline

02

### Add Application Code

Create a simple calculator app to test

03

### Write Test Files

Add passing and failing tests to demonstrate CI

04

### Configure GitHub Actions

Set up automated testing workflow

05

### Enable Branch Protection

Require CI checks before merging

**Training Wheels:** We'll use GitHub's web interface today for pedagogy. In future sessions, you'll work locally with Git commands.

Make sure you're logged into GitHub and ready to follow along. Type "ready" in chat when you're set!

# Create Your Repository

## Step-by-Step

Click the **+** icon in GitHub's top-right corner

Select **New repository**

Name it: `w5d1-ci-cd`

Choose **Public** or **Private**

**Do not** add README, .gitignore, or license

Click **Create repository**

Confirm in chat once you see your empty repository!

**Important:** Keep the repository empty for now.
We'll add files one by one to understand each component.

# Create Application Code

## Add app/calculator.py

Click **Add file** → **Create new file** in your repository. Type `app/calculator.py` as the filename (the forward slash creates the folder automatically).

```
def add(a: int, b: int) -> int:
"""Add two numbers together."""
return a + bdef subtract(a: int, b: int) -> int:

"""Subtract b from a."""
return a - bdef safe_divide(a: float, b: float) -> float:

"""Divide a by b, raising error if b is zero."""    if b == 0:
raise ZeroDivisionError("Cannot divide by zero")
return a / b
```

Click **Commit changes** to save the file.

### What This Does

- Simple calculator functions
- Type hints for clarity
- Error handling for division
- Ready to be tested by CI

# Create Passing Tests

## Add tests/test_calculator_pass.py

Click **Add file** → **Create new file** again. Name it `tests/test_calculator_pass.py`.

```python
from app.calculator import add, safe_divide

import math

def test_add_simple():
    assert add(2, 2) == 4

def test_add_negative():
    assert add(-1, 1) == 0

def test_safe_divide_regular():
    assert math.isclose(safe_divide(9, 3), 3.0)

def test_safe_divide_decimal():
    assert math.isclose(safe_divide(10, 4), 2.5)
```

Commit this file. These tests will **pass** and give us green checks.

**pytest** automatically discovers functions starting with `test_`. Assertions that pass result in green checks in CI.

# Create Failing Test (Intentionally)

## Add tests/test_calculator_fail.py

Create one more file: `tests/test_calculator_fail.py`. This test will **intentionally fail** to demonstrate CI catching bugs.

```
from app.calculator import add

def test_add_wrong():
    # This assertion is WRONG on purpose
    assert add(2, 3) == 6  # Should be 5!
```

Commit this file. We'll see CI turn red, then we'll fix it and watch it turn green.

> **Why fail on purpose?** This demonstrates CI's core value: catching bugs before they reach production. We'll fix this shortly and experience the fast feedback loop.

# Configure GitHub Actions Workflow

## Add .github/workflows/ci.yml

Create the file `.github/workflows/ci.yml` (note the dot at the start and nested folders).

```yaml
name: CI Pipeline

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

jobs:
  build-and-test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: '3.11'
      - run: pip install -r requirements.txt
      - run: pytest -v

  deploy:
    needs: build-and-test
    runs-on: ubuntu-latest
    steps:
      - run: echo "Deploying to production..."
```

Commit this file, then navigate to the **Actions** tab to watch your first CI run!

# Add Requirements File

## Create requirements.txt

One more file: `requirements.txt` at the root of your repository.

```
pytest==8.3.2
```

Commit this file. Now go to the **Actions** tab and watch your pipeline run!

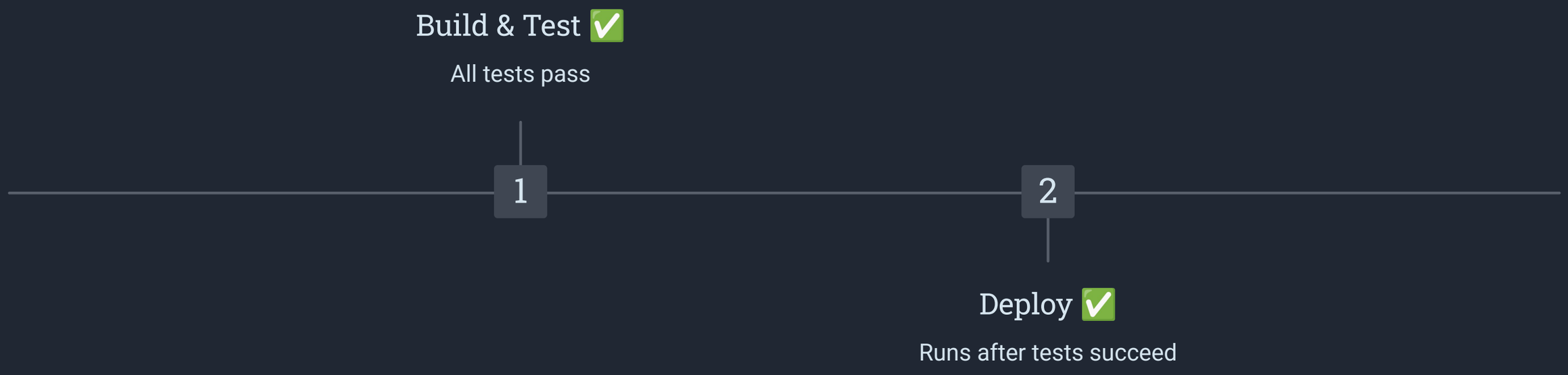
You should see a red ❌ because of our intentional failing test. This is exactly what we want—CI caught the bug!

# Fix the Failing Test

Now let's experience the CI feedback loop. Navigate to `tests/test_calculator_fail.py` in GitHub and click the edit (pencil) icon.

Change the assertion from `assert add(2, 3) == 6` to `assert add(2, 3) == 5`.

Commit the change, then go to the **Actions** tab and watch the new run. This time, you should see:




**Discussion Question:** What made the deploy job wait for the build-and-test job? Look for the `needs:` keyword in the workflow file.

## Break Time

# 10 Minutes

> **Reflection Prompt:** Type in chat one part of *your workflow* you'd love to automate—tests, linting, deploy, formatting, anything!

We'll resume at 7:45 PM. Use this time to stretch, grab water, and reflect on what we've built so far.

# Local Setup Overview

## Cloud vs Local

CI runs in the cloud automatically—you don't need a local setup for CI to work. However, running tests locally helps you catch issues *before* pushing to GitHub.

We'll cover setup for Mac, Linux, and Windows. Follow along if you want to run tests on your machine.

### Cloud CI

Runs automatically on push

### Local Dev

Optional but recommended

**Quick Poll:** Who already has Python installed locally? Type "yes" or "no" in chat.

# Check Python Installation

## Verify Python on Your System

### Mac/Linux

```
python3 --version
```

If missing, install via Homebrew:

```
brew install python
```

### Windows

```
python --version
```

If missing, download from [python.org](python.org) and check **Add to PATH** during installation.

### Expected Output

```
Python 3.11.x
```

Any version 3.10+ works. If you see Python 2.x on Mac, use `python3` instead of `python`.

### Troubleshooting

Mac: Use `python3` if `python` maps to Python 2

- Windows: Fix PATH if command not found

Verify: `python -m venv --help` should work

# Clone Repository & Setup Environment

## Get Your Code Locally

```
git clone https://github.com/YOURNAME/w5d1-ci-cd.gitcd w5d1-ci-cd
```

## Create Virtual Environment

### Mac/Linux

```
python3 -m venv .venv source .venv/bin/activate
```

### Windows (PowerShell)

```
python -m venv .venv.\.venv\Scripts\Activate.ps1
```

If blocked by execution policy:

```
Set-ExecutionPolicy -Scope Process -ExecutionPolicy RemoteSigned
```

Your shell should now show `(.venv)` at the beginning of the prompt, indicating the virtual environment is active.

# Install Dependencies & Run Tests

## Final Local Setup Steps

```
python -m pip install --upgrade pip pip install -r requirements.txt pytest -v
```

**2**

Tests Passed

If you fixed the failing test

**0**

Tests Failed

All green locally!

If you see `1 failed, 1 passed`, you haven't fixed the test yet. Go back to GitHub and update `test_calculator_fail.py`.

**Debrief:** Who got all green locally? Who hit a snag? Type in chat and we'll address common issues.

# Breakout 1: First PR + Matrix Testing

## Practice Branching & CI on Pull Requests

01

### Create a Branch

In GitHub UI, create a new branch called `feature/matrix-test`

0

2

### Edit Workflow File

Add a matrix strategy to test Python 3.10 and 3.11

0

3

### Open Pull Request

Create PR from your branch to main

0

4

### Watch CI Run

Observe parallel test runs for both Python versions

```
jobs:  build-and-test:    runs-on: ubuntu-latest    strategy:      matrix:        python-version:
['3.10', '3.11']    steps:      - uses: actions/setup-python@v5        with:
python-version: ${{ matrix.python-version }}
```

# Branch Protection Rules

## Enforce CI Checks Before Merge

### Navigate to Settings

Go to your repository → Settings → Branches

### Add Branch Protection Rule

Click "Add rule" for the main branch

### Require Status Checks

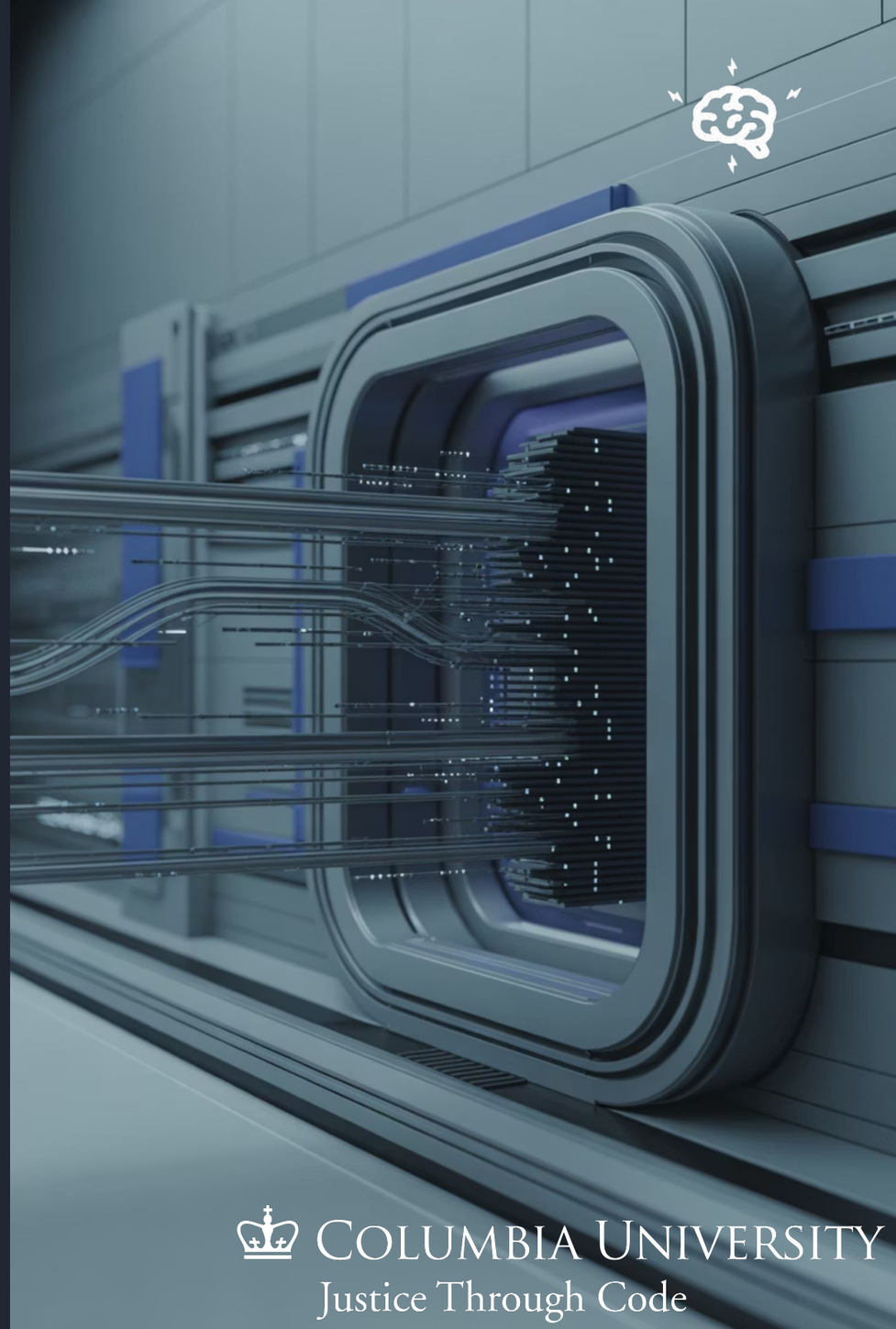
Enable "Require status checks to pass before merging"

### Save Protection

Confirm and save the rule

> **Critical Question:** What happens if CI is red? (Expected: Merge is blocked—you physically cannot merge broken code into main)

This is how professional teams keep main always deployable. No exceptions, no overrides.



# Breakout 2: Debug Common CI Failures

## Practice Fixing Realistic Pipeline Issues

### Card A: YAML Indent Error

Intentionally break the indentation in your workflow file. Watch CI turn red, then fix the spacing and watch it turn green.

### Card B: Branch Mismatch

Change the workflow to trigger on a different branch name. Notice CI doesn't run. Fix the branch name and push again.

### Card C: Test Failure

Break an assertion in your test file. Watch CI catch it. Fix the assertion and confirm green checks return.

**Choose one card, break your pipeline, and fix it.** This hands-on practice builds confidence in reading CI logs and debugging failures quickly.

**Debrief Question:** What helped you fix the issue fastest? (Expected: error messages in logs, consistent pipeline behavior, clear feedback)

# Assignment: Green Checks Only

## Post-Class Deliverables (30 minutes)

| | |
|---|---|
| 1 | **Repository Link**<br>Submit your GitHub repo URL with working CI/CD pipeline |
| 2 | **Failing Run Screenshot**<br>Capture the red ❌ from your intentional test failure |
| 3 | **Passing Run Screenshot**<br>Capture the green ✅ after fixing the test |
| 4 | **README Reflection**<br>Write 3-5 sentences on what you learned about CI/CD |
| 5 | **Branch Protection Enabled**<br>Confirm status checks are required before merging |

Assignment link is in the course portal. Due before W5D2. Recording will be posted within 2 hours if you need to review anything.

> **Remember:** CI/CD = Confidence + Speed. You now have the tools to ship code safely and frequently. Automate what hurts, then iterate.

## Tomorrow: W5D2 Decision Trees & Logic Flow

You'll build on today's CI/CD foundation by exploring software architecture patterns. You'll learn Domain-Driven Design and Hexagonal Architecture—strategies that separate business logic from technical infrastructure. Through refactoring exercises, you'll see how well-structured code makes your CI pipelines faster and your tests more reliable. We're moving from automation to professional software design.

# Questions?

Columbia University
Justice Through Code