Module 4: Intelligent Refactoring and CI/CD, Lecture 4.4: Shipping with Confidence

Hands-On: Build a Production CI/CD Pipeline in 3 Prompts

Overview

This demo proves three claims from the lecture in real time:

- CI/CD pipelines are infrastructure-as-code — and AI writes YAML extremely well when you give it structure
- A production-grade pipeline is not just “tests on commit” — it is caching, parallelism, security scanning, coverage gates, and gated deploys
- When the pipeline breaks, the AI debugging loop turns a 30-minute hunt into a 2-minute fix

You will go from an empty `.github/workflows/` directory to a green, multi-environment, security-scanned pipeline using exactly three prompts in the same AI chat session. The student lab extends this with a manual rollback workflow and Slack notifications.

Before You Start

- Open your AI tool of choice: Claude.ai, Cursor, or GitHub Copilot Chat
- Get your own copy of the sample app — go to `github.com/AI-Native-Engineer-Course/course-sample-api`, click **Use this template** → create the repo under your own account, then `git clone` it locally
- Set up the project locally: `python -m venv .venv` → activate it → `pip install -r requirements-dev.txt` → run `pytest` and confirm **4 passed** before continuing. The pipeline needs a real green test suite to run against — if pytest fails locally, the AI-generated workflow will fail in CI too.
- Keep the four-part prompt template visible on a second monitor throughout
- Do NOT start a new chat between prompts — session continuity is the whole point

> **Pre-flight Setup (10 minutes — do BEFORE the lesson starts)**
> If you have never set up the sample app before, follow these six steps exactly:
>
> 1. **Template the repo:** Go to `github.com/AI-Native-Engineer-Course/course-sample-api` → click **Use this template** → **Create a new repository** → name it `course-sample-api`, set it **Public**, create under your personal account.

2. **Clone locally:** `git clone https://github.com/<your-username>/course-sample-api.git` then `cd course-sample-api`.

3. **Create a virtualenv (Windows users use cmd or PowerShell):** `python -m venv .venv`, then activate — macOS/Linux: `source .venv/bin/activate`; Windows cmd: `.venv\Scripts\activate.bat`.

4. **Install dev deps:** `pip install -r requirements-dev.txt` — this installs FastAPI, pytest, ruff, bandit, and the rest.

5. **Verify the suite passes:** `pytest` — you must see **4 passed** before continuing. If you don't, raise your hand — the rest of the lesson will not work.

6. **Confirm git remote:** `git remote -v` — `origin` should point at your fork, not the course org. You will push the AI-generated workflow file here in a moment.

Remember

Every CI/CD prompt follows the same four parts. We will prove it by building a production pipeline from scratch — base CI first, then security and coverage, then multi-environment deploy — all in one session. Watch the pattern repeat across all three prompts.

During Hands-On — Keep Visible if Possible

The four-part CI/CD prompt template:

```
1. ROLE          — who the AI is acting as (e.g. senior DevOps engineer)
2. TRIGGER       — when the workflow runs (push, PR, schedule, manual)
3. JOBS          — what runs (lint, test, build, scan, deploy)
4. REQUIREMENTS  — runner OS, caching, parallelism, summary, gates
```

Prompt 1 — Base CI Pipeline (Lint + Test + Build)

What to Learn

Here's the four-part prompt in action. Tell the AI exactly what role, what trigger, what jobs, and what runner-level requirements — and critically, tell it to use caching and parallelism from the start. We are not bolting performance on later.

Paste this into the AI chat:

```
ROLE: You are a senior DevOps engineer.
```

```
TRIGGER: Run on push to any branch and on pull request to main.

JOBS (run in parallel — no `needs` dependencies between them):
  1. lint   — Run `ruff check .` against the whole repo.
              Fail the build on any finding.
  2. test   — Run `pytest` from the repo root.
              Fail the build if any test fails.
  3. build  — Build the Docker image from the repo's Dockerfile.
              Tag locally; do NOT push.

REQUIREMENTS:
  - ubuntu-latest runner for every job
  - Python 3.11
  - Cache pip dependencies using the hash of
    requirements-dev.txt as the cache key
  - Display a GitHub Actions job summary table with
    pass/fail per job
  - Add a one-line YAML comment to every non-obvious step

BEHAVIOR:
  - Output a single complete .github/workflows/ci.yml file
  - No prose before or after the YAML — just the file
```

> **What you should get**
> A complete ci.yml with three parallel job lanes, pip caching keyed on requirements-dev.txt, inline comments, and a job-summary step. Paste it into .github/workflows/ci.yml, commit, push — and watch the pipeline go green in the Actions tab in 60–90 seconds.

Then commit and push:

```
mkdir -p .github/workflows
# paste the AI's output into .github/workflows/ci.yml
git add .github/workflows/ci.yml
git commit -m "ci: add base GitHub Actions workflow (AI-generated)"
git push
```

Switch to the **Actions** tab in your GitHub repo. Refresh. You should see the workflow run appear within seconds and the three lanes — lint, test, build — start in parallel. Total run time on a clean cache: ~90 seconds. On the second run: ~25 seconds.

Prompt 2 — Hardened CI (Security + Coverage)

> **Remember**
> A lint-and-test pipeline is good. A pipeline with security scanning and coverage gating is production-ready. Notice the prompt below tells the AI to MODIFY the existing workflow, not regenerate it from scratch — that protects the inline comments and structure we just got right.

Continue in the SAME chat session:

```
Modify the ci.yml from your last response.

ADD a fourth parallel job named `security` that:
  1. Runs `pip-audit -r requirements.txt`
     to flag dependency CVEs
  2. Runs `bandit -r app/` for Python SAST
  3. Runs the gitleaks Action to scan for committed secrets
  4. Fails the build on any HIGH or CRITICAL finding

ALSO modify the existing `test` job to:
  - Run pytest with coverage:
    `pytest --cov=app --cov-report=xml --cov-report=term`
  - Fail the build if branch coverage drops below 80%
  - Upload coverage.xml as a workflow artifact

BEHAVIOR:
  - Output the complete updated ci.yml — not just a diff
  - Preserve every existing inline comment
  - Add inline comments to every new step too
```

> **What you should get**
> An updated ci.yml with four parallel lanes (lint / test / build / security), pytest now invoked with --cov flags, a coverage gate, and an artifact upload. The job summary will now show four green checks instead of three. Commit, push, and confirm in the Actions tab.

Commit and push:

```
git add .github/workflows/ci.yml
git commit -m "ci: add security scanning + coverage gate"
git push
```

Tip: *If bandit or pip-audit flags a false positive that blocks the demo, add* `continue-on-error: true` *to the offending step and narrate aloud: "In production you triage these findings; for the demo we want to keep moving — the point is the tool flagged something, which is the win."*

Prompt 3 — Multi-Environment Deploy with Human Approval

> **What to Learn**
> Now the production gate. Code that passes CI should deploy to staging automatically — but reaching production deserves a human decision. This last prompt produces a separate deploy workflow that uses GitHub Environments for the approval gate.

Same session — third and final prompt:

```
Generate a SECOND workflow file: .github/workflows/deploy.yml.

TRIGGER: workflow_run — fires only when ci.yml completes
successfully on the main branch.

JOBS:
  1. deploy-staging:
     - Runs automatically (no approval gate)
     - environment: staging
     - Steps: echo placeholder for build push + deploy
  2. deploy-production:
     - Requires manual approval via
       GitHub Environments protection rules
     - environment: production
     - needs: [deploy-staging]
     - Steps: echo placeholder for deploy command

REQUIREMENTS:
  - Add an inline YAML comment explaining each of:
    workflow_run, environment, needs
  - Both deploy jobs run on ubuntu-latest
  - Use ${{ secrets.DEPLOY_TOKEN }} placeholders where
    real credentials would go

BEHAVIOR: Output the complete deploy.yml file.
```

> **What you should get**
> A second workflow file that triggers off ci.yml's success. Once committed, you will configure the production environment under Repo → Settings → Environments → New environment → name: production → check Required reviewers and add 1–2 reviewers. The next push to main will deploy to staging automatically and then PAUSE at the production gate waiting for human approval.

Commit, push, and configure the environment:

```
git add .github/workflows/deploy.yml
git commit -m "ci: add multi-env deploy with manual prod gate"
git push

# Then in the GitHub UI:
#   Settings → Environments → New environment → name: production
#   Check "Required reviewers" and add 1-2 teammates
#   (Optional) Add a "Wait timer" of 5 minutes as a cooling-off period
```

Bonus — When the Pipeline Breaks: The AI Debug Loop

> **What to Learn**
> Every CI/CD pipeline eventually fails. Instead of hunting through stack traces and YAML reference docs, paste the failed job log into AI with a structured debug prompt. The 30-minute hunt collapses into a 2-minute fix. Practice this NOW so you have the muscle memory when it matters.

Introduce a deliberate failure: open ci.yml, change `python-version: '3.11'` to `python-version: '3.99'`, commit and push. Watch the run go red in the Actions tab. Open the failed job, copy the full log, then paste this prompt into AI:

```
You are a senior DevOps engineer. A GitHub Actions job in
this repository just failed. The full log is below.

TASK: Identify:
  1. The root cause (why it failed, not just what failed)
  2. The specific YAML line(s) or config to change
  3. The corrected snippet, ready to paste back into ci.yml

Do NOT speculate about anything not visible in the log.

[paste the failed job log here]
```

> **What you should get**
> A diagnosis pointing to the offending line, an explanation of the failure mode (invalid version, missing secret, version conflict, timeout, flaky test, etc.), and the corrected YAML snippet. Apply the fix, push, watch the pipeline go green. Total time from red to green: under 2 minutes.

Wrap-Up — The Pattern, The Phase 1 Capstone Handoff

> **Remember**
> Three prompts. One session. We got a parallel base CI, shift-left security with a coverage gate, and a multi-environment deploy with a human approval gate before production. The whole pipeline is documented inline because we asked for comments in the BEHAVIOR block. That is contract-first thinking applied to infrastructure.

Key takeaways:

- Four-part prompt every time: ROLE, TRIGGER, JOBS, REQUIREMENTS
- Caching and parallelism go in from prompt one — never bolt them on later
- Shift-left: SAST, dependency scans, and secret scans run on every commit, not annually
- Automation handles what it can reliably handle; the production gate stays human
- When it breaks, paste the full log into AI with a structured debug prompt — 30-minute hunt becomes a 2-minute fix

Phase 1 Capstone handoff:

This hands-on closes Module 4 and Phase 1. For the capstone, take the legacy app from Lesson 1, run the diagnostic, refactor the top two priority issues from Lesson 2, hit 80% test coverage from Lesson 3, and ship it through the pipeline you just built. Submit the repo link, the `health_report.md`, the refactor PR, a coverage screenshot, and a 60-second video walkthrough. See you in Phase 2.