PROGRAMMING

CI/CD Pipelines

GitHub Actions Fundamentals · Building & Testing

Secrets & Environments · Docker in CI/CD

Deployment Strategies · Continuous Deployment

Capstone: A Full Real Pipeline

Philip Osztromok

Generated with Claude

Table of Contents

CI/CD Pipelines — 9 Chapters

🚀 Why CI/CD? The Problem It Solves

Every language course in this curriculum already taught you how to write tests — Go's testing package, Jest, RSpec, PHPUnit. This course isn't about writing more tests. It's about the fact that in most shops, actually running those tests before merging or deploying is still a manual step someone has to remember — and automating that step, and everything that follows it, is what CI/CD actually is.

The Manual Deployment Problem

A typical pre-CI/CD release process: a developer runs tests locally (or skips them "just this once" under deadline pressure), manually builds the app, manually copies files to a server — or worse, edits files directly on production — then manually restarts the service. Every one of these steps is a place for human error, inconsistency between environments ("works on my machine"), and forgotten steps exactly when there's least time to catch them.

📖 CI, CD, and Continuous Deployment

These three terms get blurred together in casual conversation, but the distinctions matter for understanding what a given pipeline actually promises:

Term	What It Actually Means
Continuous Integration (CI)	Every code change is automatically built and tested — catching integration problems early, not at release time
Continuous Delivery (CD)	Every passing build is automatically prepared for release (built, tested, packaged) — but a human still approves the actual deployment
Continuous Deployment	Goes one step further — every passing build deploys straight to production with no human approval step at all

Before and After

Manual Process

Remember to run tests, manually build, manually copy files to a server, manually restart — every step a chance to forget something under pressure.

Automated Pipeline

Push code → tests run automatically → build happens automatically → deployment proceeds (or waits for approval) automatically — no step depends on someone remembering it.

Why Automating Testing Specifically Matters Most

A test suite that only runs when a developer remembers to run it locally provides much weaker guarantees than one that runs automatically and **blocks a merge or deploy on failure**. This is the single most valuable, foundational piece of CI/CD — even before touching deployment at all.

What You Already Know

Writing tests themselves — covered across the Go, Node.js, Express, and TypeScript courses. This course assumes that skill and builds automation around it.

Where This Course Ends Up

By Chapter 9's capstone: a full pipeline that builds, tests, containerizes, and deploys an app to staging and production with an approval gate.

💻 Coding Challenges

Challenge 1: Classify Three Pipeline Descriptions

For each of (a) "every PR runs the test suite automatically," (b) "every passing build is packaged and a team lead clicks Deploy," and (c) "every merge to main goes live within minutes with no human involved," identify whether it describes CI, Continuous Delivery, or Continuous Deployment.

Goal: Practice distinguishing the three terms in realistic descriptions, not just definitions.

→ Solution

Challenge 2: Identify the Manual-Process Failure Mode

A team's release process requires a developer to manually SSH into a production server and run a deploy script. Identify at least two distinct failure modes this manual step introduces, referencing this chapter's own examples.

Goal: Practice recognizing concrete risks in a manual process, not just labeling it "risky."

→ Solution

Challenge 3: Decide Whether Continuous Deployment Fits

A small internal tool used only by 5 employees has a solid automated test suite. Discuss whether Continuous Deployment (no human approval before production) is a reasonable choice here, using this chapter's definitions.

Goal: Practice reasoning about when the strongest form of automation is actually appropriate, not just "the most advanced" option.

→ Solution

⚠️ Gotcha: "We Have CI" Doesn't Mean "We're Safe"

CI only automates *running* whatever tests already exist — it does nothing to improve the quality or coverage of a weak test suite. A pipeline that runs zero or trivial tests and reports "passing" gives a false sense of safety that can be worse than knowing tests aren't automated at all, since a green checkmark now implies a confidence that isn't actually earned. CI/CD automates test *execution*, not test *quality* — the two are easy to conflate, and conflating them is a genuinely common, costly mistake.

🎯 What's Next

With the terminology and motivation established, the next chapter breaks down the actual structure every pipeline shares: **Anatomy of a Pipeline** — stages, triggers, jobs vs steps, and a pipeline as a dependency graph.

🧬 Anatomy of a Pipeline

Chapter 1 established why CI/CD exists. This chapter establishes the structure every pipeline shares, regardless of which specific tool implements it — GitHub Actions, GitLab CI, CircleCI, Jenkins — so Chapter 3's GitHub Actions specifics land on solid conceptual ground.

Stages

Stage	What Happens
Build	Compile/transpile code, install dependencies
Test	Run the automated test suites from Chapter 1
Deploy	Ship the result somewhere

Not every pipeline has all three, and some split further (a lint stage, a security-scan stage) — but build → test → deploy is the near-universal backbone.

⚡ Triggers

What causes a pipeline to run — and different triggers commonly map to different stage subsets:

Trigger	Typical Stages Run
Pull request	Build + test only — nothing is merged yet, so nothing deploys
Push to main	Build + test + deploy — a merge has actually happened
Schedule (cron)	Whatever a periodic job needs — e.g. a nightly full test run
Manual (`workflow_dispatch`)	Whatever a human explicitly requests — e.g. deploying a specific version on demand

🧱 Jobs vs Steps

A Job

A unit of work running on its own dedicated runner — jobs can run in **parallel** with each other unless a dependency says otherwise.

A Step

One individual command or action *within* a job, running sequentially on the same machine, sharing that job's filesystem.

This distinction matters for both performance (parallel jobs run the whole pipeline faster) and correctness: steps within one job can rely on an earlier step's output being on disk — a *different* job cannot, unless that data is explicitly passed between them.

A Pipeline as a Dependency Graph

Jobs can depend on other jobs completing first, making a pipeline a directed graph, not a flat list:

A Simple Graph: Build → [Lint, Test in Parallel] → Deploy

- **build** — runs first, no dependencies
- **lint** — depends only on `build`; runs in parallel with `test`
- **test** — depends only on `build`; runs in parallel with `lint`
- **deploy** — depends on *both* `lint` and `test` finishing successfully

One Long Job vs Parallel Jobs

> #### Everything in One Job
>
> Lint, then test, then build artifacts — all crammed into one linear sequence of steps, running one after another with no parallelism at all.

> #### Correctly Split Jobs
>
> Lint and test identified as genuinely independent work, run as separate parallel jobs — the pipeline finishes in whichever one takes longer, not the sum of both.

💻 Coding Challenges

Challenge 1: Choose Stages for Two Triggers

For a pipeline triggered by (a) a pull request and (b) a push to the `main` branch, list which stages from this chapter's table should run for each, and explain the one key difference.

Goal: Practice mapping triggers to appropriate stage subsets.

→ Solution

Challenge 2: Draw a Dependency Graph

A pipeline has four jobs: `build` (no dependencies), `unit-tests` (depends on `build`), `integration-tests` (depends on `build`), and `deploy` (depends on both test jobs). Describe which jobs can run in parallel and which must wait.

Goal: Practice reasoning about a dependency graph with more than one parallel branch.

→ Solution

Challenge 3: Identify a Jobs-vs-Steps Mistake

A pipeline has a `build` job that compiles a binary to `./dist/app`, and a separate `deploy` job that tries to copy `./dist/app` to a server — but the deploy job fails with "file not found." Explain why, using this chapter's jobs-vs-steps distinction.

Goal: Practice diagnosing the exact mistake this chapter's tip box describes.

→ Solution

⚠️ Gotcha: Assuming File State Carries Over Between Jobs

It's a very common early mistake to assume a later job can simply read a file an earlier job produced, the way a later *step* within the same job reliably can. In reality, each job typically runs on a fresh, isolated machine or container with no automatic file sharing between jobs — a "build" job's compiled output is gone the moment that job's runner shuts down, unless it's explicitly passed forward as an "artifact" (uploaded in one job, downloaded in another). Forgetting this produces exactly the confusing experience of a deploy job failing with "file not found," even though the build job clearly reported success moments earlier.

🎯 What's Next

With the universal structure established, the next chapter makes it concrete with a real tool: **GitHub Actions Fundamentals** — workflow YAML syntax, the actions marketplace, and a first working workflow.

⚙️ GitHub Actions Fundamentals

Chapter 2 established the universal anatomy every pipeline shares. This chapter makes it concrete with a real, specific tool — GitHub Actions — where every concept from Chapter 2 maps directly onto actual YAML syntax.

Where Workflows Live

Workflow files sit at `.github/workflows/*.yml` in a repository — each file is one workflow, and a repo can have several.

A Quick YAML Primer

Key-value pairs, nesting via indentation, lists via a `-` prefix — YAML is whitespace-sensitive and never uses tabs. Indentation mistakes are a very common early source of confusing workflow errors.

The Core Structure

YAML Key	Maps to Chapter 2's Concept
`on:`	Triggers — `push`, `pull_request`, `schedule`, `workflow_dispatch`
`jobs:`	A map of job-id → job definition — Chapter 2's "jobs"
`runs-on:`	Which OS/runner image a job uses
`steps:`	A list of commands/actions within a job — Chapter 2's "steps"

Two Kinds of Steps

`run:` Steps

Execute a shell command directly — `npm install`, `npm test`.

`uses:` Steps

Invoke a reusable, versioned action from the marketplace — `actions/checkout@v4`, `actions/setup-node@v4`.

`actions/checkout` is almost always the very first step in any real job — Chapter 2's "isolated machine" point resurfaces concretely here: a fresh runner starts with a completely empty filesystem. Without checking out the repo, there's no code to build or test at all.

🛒 The Actions Marketplace

Action	Purpose
`actions/checkout`	Pulls the repo's code onto the runner
`actions/setup-node` / `setup-go` / `setup-python`	Installs a specific language runtime version
`actions/cache`	Caches dependencies between runs
`actions/upload-artifact` / `download-artifact`	Passes files between jobs — Chapter 2's artifact fix

Hand-Rolled vs Marketplace Action

> **Hand-Rolled**
>
> Writing raw shell commands to clone the repo and install a specific Node version — more to get wrong, no community maintenance.

> `actions/checkout` + `actions/setup-node`
>
> Two well-tested, versioned, widely-used actions — less code, fewer edge cases to handle yourself.

🎬 A First Complete Workflow

Tying directly back to Chapter 1's framing — automating the tests already covered in the Node.js/Express courses:

`.github/workflows/ci.yml`

```
name: CI

on: [push, pull_request]

jobs:
  build-and-test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: '20'
      - run: npm install
      - run: npm test
```

💻 Coding Challenges

Challenge 1: Write a Workflow for a Go Project

Write a complete GitHub Actions workflow that triggers on push, checks out the repo, sets up Go 1.22 via `actions/setup-go`, and runs `go test ./...`.

Goal: Practice adapting this chapter's Node.js example to a different language's marketplace action.

→ Solution

Challenge 2: Identify run vs uses

For each of (a) `actions/checkout@v4`, (b) `npm run build`, and (c) `actions/setup-python@v5`, state whether it belongs under `run:` or `uses:`, and why.

Goal: Practice distinguishing the two step types quickly and correctly.

→ Solution

Challenge 3: Diagnose a Missing Checkout

A workflow's first step is `run: npm install` (no `actions/checkout` beforehand), and it fails immediately with an error about a missing `package.json`. Explain exactly why, referencing this chapter's and Chapter 2's material.

Goal: Practice diagnosing the single most common first-workflow mistake.

→ Solution

⚠️ Gotcha: Forgetting `actions/checkout` as the First Step

It's easy to assume a runner already has the repo's code, the way a local development machine does — it doesn't. Chapter 2's "fresh, isolated machine" point applies here concretely: without `actions/checkout@v4` as the very first step, the runner has no `package.json`, no source files, nothing at all — every subsequent step (`npm install`, `npm test`) fails immediately, not because those commands are wrong, but because there's simply no repository present to run them against. Make `actions/checkout` the first step in any job that needs the repo's own files, every time.

🎯 What's Next

With a first working workflow in hand, the next chapter goes deeper on the build/test stages themselves: **Building & Testing in CI** — matrix builds, dependency caching, and running real test suites inside a workflow.

🧪 Building & Testing in CI

Chapter 3 built one working workflow. This chapter goes deeper on the build/test stages specifically — making them faster with caching and more thorough with matrix builds, the two most common practical upgrades teams make to a basic pipeline.

Matrix Builds

Running the *same* job definition across multiple combinations of variables — Node 18/20/22, or multiple operating systems — automatically, in parallel. This directly extends Chapter 2's "jobs run in parallel" concept: a matrix job is really N parallel jobs generated from one definition.

```
jobs:
 test:
  strategy:
   matrix:
    node-version: [18, 20, 22]
  runs-on: ubuntu-latest
  steps:
   - uses: actions/checkout@v4
   - uses: actions/setup-node@v4
    with:
     node-version: ${{ matrix.node-version }}
   - run: npm ci
   - run: npm test
```

This catches "works on Node 20 but breaks on Node 18" bugs before they reach users. A library published for others to use needs to actually work across whatever version range it claims to support — testing only the developer's own local version silently assumes every user matches that exact version, which is rarely true.

📦 Caching Dependencies

Without caching, every single run re-downloads every dependency from scratch — even though nothing changed most of the time. Dependencies get cached, keyed by a hash of the lockfile's contents:

```
- uses: actions/setup-node@v4
  with:
    node-version: '20'
    cache: 'npm'  # keyed internally off package-lock.json's hash
```

Why Hash the Lockfile

A cache key must change when dependencies change — hashing the lockfile's contents guarantees the cache correctly invalidates the moment a dependency is added, removed, or updated.

The Speed Payoff

A cache hit skips re-downloading unchanged dependencies entirely — often the single biggest speed improvement available to a basic pipeline.

Running Existing Test Suites

Nothing new to learn here conceptually — it's about correctly invoking whatever test command the Go, Jest, or other course already taught (`go test ./...`, `npm test`) as a CI step, and letting that step's own exit code determine pass or fail.

🚦 Failing the Build on Test Failure

This is the actual enforcement mechanism that makes CI meaningful — Chapter 1's "blocks a merge on failure" point, made concrete. A step returning a non-zero exit code automatically fails the whole job. Most test runners already exit non-zero on any failing test by default — "failing correctly" is usually automatic, not something to manually configure.

No Caching/Matrix vs Both

Basic Workflow

Every run re-downloads everything from scratch, and only tests one Node version — slow, and blind to real compatibility bugs.

Cached + Matrix

Dependency install is nearly instant on a cache hit; three Node versions tested in parallel catch bugs a single-version run would miss entirely.

💻 Coding Challenges

Challenge 1: Add a Python Version Matrix

Write a job's `strategy: matrix:` block testing a Python project across versions 3.10, 3.11, and 3.12.

Goal: Practice writing a matrix block for a language other than this chapter's Node.js example.

→ Solution

Challenge 2: Explain What Breaks a Bad Cache Key

A workflow caches `node_modules` using the static key `"npm-cache"` (not derived from the lockfile). Explain what happens the next time a dependency is added to `package.json`, and why it's a problem.

Goal: Practice reasoning about the exact failure mode this chapter's tip box describes.

→ Solution

Challenge 3: Spot the Swallowed Exit Code

A workflow step is written as `run: npm test | tee test-output.log`. Explain why this can cause a pipeline to report success even when tests actually failed, and provide a fix.

Goal: Practice recognizing a subtle way the "fail on non-zero exit code" mechanism gets silently defeated.

→ Solution

⚠️ Gotcha: Caching With a Static Key

Using a fixed cache key like `"node-modules-cache"` instead of a hash of the lockfile's contents silently reuses a stale cache even after dependencies have actually changed — nothing about the key itself changed to signal "this cache is now invalid." The result: a pipeline that passes using *old*, cached dependencies instead of the new ones a developer just added, potentially masking a real bug that only manifests with the correct, up-to-date dependencies, or missing a genuinely needed new package entirely. Always key caches off a hash of the lockfile's actual contents, never a fixed string.

🎯 What's Next

With builds and tests running fast and thoroughly, the next chapter covers what a pipeline needs to keep private: **Environment Variables & Secrets** — GitHub Secrets, and never committing credentials.

🔐 Environment Variables & Secrets

Chapter 4 covered building and testing. This chapter covers something every real pipeline eventually needs — credentials — and how to handle them without ever putting them in the repository itself, extending the environment/configuration material from the Node.js and TypeScript courses.

Never Commit Credentials

The same "never expose secrets" principle running through the site's security courses applies directly here. A credential committed to git history is compromised *forever*, even if later deleted from the current file — git retains every prior commit. This is a genuinely common real-world incident pattern: automated bots scan public GitHub pushes and scrape leaked API keys within minutes of being committed.

🔑 GitHub Secrets

Where credentials actually belong instead — stored encrypted at the repo or organization level, added via **Settings → Secrets and variables → Actions**, referenced in workflows via `secrets.SECRET_NAME`. GitHub automatically redacts a secret's value if it appears in log output, replacing it with `***`.

Using a Secret in a Workflow

```
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - run: ./deploy.sh
        env:
          DATABASE_URL: ${{ secrets.DATABASE_URL }}
```

This is how a deploy step gets access to a real production credential without that credential ever touching the repo's own files.

Environment-Specific Configs

Dev, staging, and production typically need different values for the same conceptual setting — a different database URL per environment. GitHub Environments (Chapter 8) let secrets be scoped per environment, so "the production `DATABASE_URL`" and "the staging `DATABASE_URL`" can share a name but hold different values depending on which environment a job is deploying to.

🛡️ Secret Scanning as Defense in Depth

GitHub's built-in secret scanning automatically detects patterns matching known credential formats (AWS keys, common API token shapes) if accidentally pushed, and can alert or block the push. This is a safety net, not a replacement for keeping secrets out of the repo in the first place — defense in depth means multiple independent layers, not relying on just one.

Plain Env Var vs Real Secret

A non-sensitive value like `NODE_ENV` can be hardcoded directly in the workflow. A real credential must always go through `secrets.*`.

Local Development Equivalent

A local `.env` file plus a `.gitignore` entry for it is the local-machine version of the same principle — credentials live outside version control everywhere, not just in CI.

Hardcoded vs Secret Reference

> **Hardcoded in YAML**
>
> Visible to anyone with repo read access, and permanently committed to git history — recoverable forever, even after deletion.

> `secrets.DATABASE_URL`
>
> Encrypted, redacted in logs, and never present in the repo's files or git history at all.

💻 Coding Challenges

Challenge 1: Fix a Hardcoded API Key

A workflow step reads `run: curl -H "Authorization: Bearer sk_live_abc123" https://api.example.com`. Rewrite it using a GitHub Secret.

Goal: Practice the single most fundamental secrets fix.

→ Solution

Challenge 2: Explain Why Deleting a Committed Secret Isn't Enough

A developer commits an API key by mistake, notices immediately, and deletes it in the very next commit. Explain why this key should still be treated as compromised.

Goal: Practice reasoning about git history's permanence, not just the current file state.

→ Solution

Challenge 3: Choose Which Values Need `secrets.*`

For each of (a) `NODE_ENV: production`, (b) a Stripe secret key, and (c) a public API base URL like `https://api.example.com`, decide whether it needs to go through `secrets.*` or can be hardcoded in the workflow.

Goal: Practice distinguishing genuinely sensitive values from ordinary configuration.

→ Solution

⚠️ Gotcha: Accidentally Logging a Secret Anyway

GitHub's automatic redaction only works for the exact known `secrets.*` value it's tracking — if a script explicitly echoes it (`run: echo "Connecting with $DATABASE_URL"` for debugging), or constructs a similar-looking string from parts of the secret in a way redaction doesn't recognize, the value can leak into logs anyway. Treat automatic redaction as a backstop, not a guarantee — the safest practice is to never intentionally print a secret's value at all, even temporarily for debugging.

🎯 What's Next

With credentials handled safely, the next chapter brings containers into the pipeline: **Docker in CI/CD** — building and pushing an image as a pipeline step, reusing the multi-stage build pattern from earlier deployment chapters.

🐳 Docker in CI/CD

Chapter 5 covered secrets. This chapter brings containers into the pipeline — reusing the multi-stage build pattern already covered in Express and Node.js's deployment chapters, now built for real inside CI rather than just conceptually.

Why Containerize in CI

A Docker image packages the app together with its exact runtime environment — eliminating "works on my machine" at the *deployment* level, not just development. The same image that passed CI's tests is the exact image that gets deployed — no rebuild-and-hope-it's-identical step in between.

Building an Image as a Pipeline Step

```
docker build -t myapp:${{ github.sha }} .
```

Tagging with the git commit SHA — not just `latest` — means always being able to identify exactly which commit produced a given running container, critical for debugging and for the rollback techniques Chapter 7 builds on.

Multi-Stage Builds, Revisited

A "builder" stage with full dev dependencies compiles the app; a final slim stage copies only the compiled output, discarding the builder stage's bulk — the same pattern from Express/Node's deployment chapters, now reused because CI is often where these images actually get built for real.

📤 Pushing to a Container Registry

```
- run: echo "${{ secrets.REGISTRY_TOKEN }}" | docker login ghcr.io -u ${{ github.actor }} --password-stdin
- run: docker push ghcr.io/myorg/myapp:${{ github.sha }}
```

Docker Hub or GitHub Container Registry (`ghcr.io`) — authentication uses a registry token, Chapter 5's secrets material applied directly: never hardcode registry credentials. The registry becomes the source of truth for which built images actually exist and are available to deploy.

Tagging Strategy

SHA tags for traceability, semantic version tags for human-readable releases — both have a place; `latest` alone has serious limits.

Job Dependency Enforces Order

Chapter 2's dependency graph applies directly: the image-build job depends on the test job passing — a broken build never gets containerized or pushed at all.

Build-Test-Then-Build-Push, With Job Dependency

```
jobs:
 test:
  runs-on: ubuntu-latest
  steps:
   - uses: actions/checkout@v4
   - run: npm ci
   - run: npm test

 build-and-push:
  needs: test  # only runs if the test job succeeds
  runs-on: ubuntu-latest
  steps:
   - uses: actions/checkout@v4
   - run: docker build -t ghcr.io/myorg/myapp:${{ github.sha }} .
   - run: echo "${{ secrets.REGISTRY_TOKEN }}" | docker login ghcr.io -u ${{ github.actor }} --password-stdin
   - run: docker push ghcr.io/myorg/myapp:${{ github.sha }}
```

Rebuild at Deploy Time vs Build Once in CI

Rebuilding Separately

Real risk of drift — a subtly different image built at deploy time than the one CI actually tested.

Build Once, Deploy That Exact Image

The image tested in CI is the literal artifact deployed everywhere after — no rebuild, no drift.

💻 Coding Challenges

Challenge 1: Add a Semantic Version Tag Alongside SHA

Extend the `docker build`/`push` steps from this chapter's example to also tag and push the image as `v1.4.0` (in addition to the SHA tag), assuming the version is known ahead of time.

Goal: Practice tagging an image with more than one identifier at once.

→ Solution

Challenge 2: Make the Image Job Depend on Tests

Given two separate jobs, `test` and `build-image`, with no dependency configured between them, add the correct YAML so `build-image` never runs if `test` fails.

Goal: Practice applying Chapter 2's job-dependency mechanism to this chapter's specific use case.

→ Solution

Challenge 3: Diagnose a Rollback Failure

A production incident requires rolling back to "the version from yesterday," but every image was tagged only as `latest`, and today's build already overwrote it. Explain why the team can't identify or pull back yesterday's exact image.

Goal: Practice connecting this chapter's tagging guidance to a real operational failure.

→ Solution

⚠️ Gotcha: Tagging Images Only With `latest`

Tagging every build only as `latest` makes it impossible to know which specific commit or version is actually running in production at any given moment — `latest` gets overwritten every single build. If a bad deploy needs rolling back, there's no reliable way to identify or pull a specific prior version, since "latest from 10 minutes ago" and "latest right now" are indistinguishable once overwritten. Always tag with something immutable and traceable — a commit SHA, a semantic version — in addition to, or instead of, `latest`.

🎯 What's Next

With a traceable, registry-hosted image in hand, the next chapter covers how it actually reaches production: **Deployment Strategies** — rolling, blue-green, and canary deployment, and automatic rollback on health-check failure.

🚦 Deployment Strategies

Chapter 6 produced a traceable, registry-hosted image. This chapter covers how that image actually replaces what's currently running in production — without an outage, and without serving broken code to everyone at once.

The Core Problem

Simply stopping the old version and starting the new one means downtime — unacceptable for most real services. Every strategy below is a different answer to transitioning from old to new code while the service keeps serving traffic.

Rolling Deployment

Replace instances of the old version gradually — a few at a time, waiting for each batch to pass health checks before moving to the next. Simplest strategy, no extra infrastructure needed, but a bug in the new version affects a growing fraction of traffic as the rollout proceeds, and rolling back mid-rollout means dealing with a mix of both versions simultaneously for a while.

🔵🟢 Blue-Green Deployment

Maintain two complete, identical production environments — "blue" currently live, "green" idle. Deploy the new version entirely to green, test it there while blue still serves all real traffic, then switch *all* traffic to green at once. Rollback is instant: switch back to blue, which was never touched.

The Tradeoff

Requires double the infrastructure — running two full environments simultaneously, at least during the switch window. A genuinely more expensive setup.

The Payoff

Instant, guaranteed-clean rollback — the old environment was never touched at all, unlike rolling deployment's mid-rollout mixed-version state.

🐤 Canary Deployment

Deploy the new version to a small subset of real traffic first — 5%, say — monitor closely, then gradually increase (5% → 25% → 100%) if things look healthy. Directly reusing TypeScript Course 3's canary deployment chapter: the key advantage is limiting the *blast radius* of a bad deploy to a small fraction of real users before it ever reaches everyone.

A Conceptual Canary Rollout

1. Deploy new version to 5% of traffic; check health/error metrics for a set period
2. If healthy, increase to 25%; check again
3. If still healthy, increase to 100%
4. If unhealthy at *any* step, automatically roll back to the previous version immediately

Strategy	Infra Cost	Blast Radius	Rollback Speed
Rolling	Low	Grows as rollout proceeds	Moderate (mixed versions mid-rollback)
Blue-Green	High (double infra)	All-or-nothing at the switch	Instant
Canary	Moderate	Small, controlled	Fast (small fraction affected)

Blast Radius, at a Glance

Rolling

A bad deploy affects an ever-growing slice of traffic as the rollout continues — the longer it runs undetected, the worse it gets.

Canary

A bad deploy is caught while only 5% of traffic is exposed — dramatically limiting real user impact before it ever reaches everyone.

💗 Automatic Rollback on Health-Check Failure

Regardless of strategy, a robust pipeline defines a health check — a `GET /health` endpoint the new version must respond to correctly — and automatically rolls back to the previous known-good image if that check fails after deployment. This is what makes a deployment genuinely *safe*, not just automated. And automatic rollback is impossible without a traceable previous-version tag to roll back to — exactly Chapter 6's tagging gotcha, resurfacing here as a hard prerequisite.

💻 Coding Challenges

Challenge 1: Choose a Strategy for a High-Traffic Checkout Flow

An e-commerce site's checkout flow processes real payments and can't tolerate serving a broken version to a large fraction of users, even briefly. Recommend a deployment strategy and justify it using this chapter's table.

Goal: Practice matching a strategy to a genuinely high-stakes blast-radius concern.

→ Solution

Challenge 2: Explain Why Blue-Green's Rollback Is Instant

Explain specifically why blue-green deployment's rollback is faster and cleaner than rolling deployment's, referencing what happens to the "old" environment in each strategy.

Goal: Practice articulating the structural reason behind blue-green's rollback advantage, not just naming it.

→ Solution

Challenge 3: Design a Health Check

Propose what a `GET /health` endpoint should actually check for a web app backed by a database, beyond simply returning `200 OK` unconditionally, and explain why an unconditional 200 would be a weak health check.

Goal: Practice designing a health check that would actually catch a real failure.

→ Solution

⚠️ Gotcha: Reaching for Canary or Blue-Green by Default

An internal, low-traffic admin tool doesn't need canary's gradual-rollout complexity or blue-green's doubled infrastructure cost — a simple rolling deployment (or even a basic redeploy with a health check) is often the right choice for lower-stakes systems. Reaching for the most sophisticated-sounding strategy by default, rather than matching the strategy to the system's actual blast-radius and cost tradeoff, is a familiar overengineering pattern from earlier in this curriculum (GraphQL vs REST, WebSockets vs SSE) — the same judgment call applies here.

🎯 What's Next

With deployment strategies covered, the next chapter formalizes when deployment actually happens and who approves it: **Continuous Deployment & Environments** — auto-deploy vs manual approval gates, and GitHub Environments with protection rules.

🚪 Continuous Deployment & Environments

Chapter 7 covered *how* deployment happens. This chapter covers *when* it happens and who approves it — formalizing Chapter 1's CI/Delivery/Deployment spectrum into concrete GitHub Actions mechanics.

The Spectrum, Made Concrete

Chapter 1's distinction — CI (automated test) → Continuous Delivery (automated build/package, human clicks deploy) → Continuous Deployment (automated all the way, no human gate) — is implemented, in GitHub Actions, entirely through one feature.

🌍 GitHub Environments

A first-class GitHub feature (**Settings → Environments**) letting a repo define named environments — `staging`, `production` — each with its own secrets (Chapter 5's environment-specific secrets, made concrete) and its own protection rules.

Protection Rule	Effect
Required reviewers	A specific person or team must manually approve before a job targeting this environment runs — the human gate that separates Delivery from Deployment
Wait timer	A mandatory delay before the job proceeds — time to notice and cancel a bad merge
Deployment branch restrictions	Only specific branches (e.g. `main`) can deploy to this environment — preventing an accidental deploy from a feature branch

Referencing an environment in a job is a single line — `environment: production` — and that one line ties the job to both the environment's secrets and its protection rules simultaneously.

Staging vs Production: A Common Real Pattern

Auto-Deploy Staging, Gated Production

```
jobs:
 deploy-staging:
  needs: build-and-push
  runs-on: ubuntu-latest
  environment: staging  # no reviewers configured — deploys automatically
  steps:
   - run: ./deploy.sh staging

 deploy-production:
  needs: deploy-staging
  runs-on: ubuntu-latest
  environment: production  # required reviewer configured in Settings
  steps:
   - run: ./deploy.sh production
```

Push to `main` automatically deploys to **staging** — Continuous Deployment for staging, since the stakes are low and it's easy to revert. Deploying the same build to **production** requires a manual approval — Continuous Delivery for production, since the stakes are higher and a human makes the final call. This staging-auto/production-gated split is a genuinely common, sensible real-world pattern.

No Environment Concept vs Staging + Production Split

One Undifferentiated Deploy Job

Same secrets, same rules, deploys straight to production on every merge — no distinction between low-stakes and high-stakes targets at all.

Separate GitHub Environments

Staging deploys automatically; production requires a reviewer's approval — the same pipeline, different gates for different stakes.

💻 Coding Challenges

Challenge 1: Add a Wait Timer to a Production Deploy

Given the `deploy-production` job from this chapter's example, describe (conceptually, since this is configured in repo Settings, not YAML) what adding a 10-minute wait timer protection rule would change about the deployment flow.

Goal: Practice reasoning about a protection rule's practical effect, not just naming it.

→ Solution

Challenge 2: Restrict Production Deploys to `main`

Explain what deployment branch restrictions on the `production` environment would prevent, using a concrete scenario involving a feature branch.

Goal: Practice connecting a protection rule to a specific, realistic mistake it prevents.

→ Solution

Challenge 3: Diagnose an Unprotected "Production" Environment

A team names a job's environment `production` but never configures any protection rules for it in Settings. Explain what actually happens when that job runs, and why the environment's name alone provides no safety.

Goal: Practice recognizing this chapter's core gotcha in a concrete scenario.

→ Solution

⚠️ Gotcha: A "Production" Environment With No Protection Rules

Simply naming an environment "production" does not automatically make it safer or gated in any way. Without explicitly configured protection rules — required reviewers, branch restrictions, a wait timer — an environment named "production" behaves exactly like an unprotected job would, running immediately with no approval step at all. The false sense of safety comes purely from the name; the actual protection comes entirely from the rules configured for it in repo Settings, and forgetting to configure them defeats the entire point of using an Environment in the first place.

🎯 What's Next

Every piece is now in place — the final chapter is a capstone: **Building a Full Pipeline for a Real App**, combining build, test, Docker, and a staging/production deploy with an approval gate into one working pipeline.

🏁 Capstone: Building a Full Pipeline for a Real App

Every previous chapter built one piece in isolation. This capstone assembles a complete pipeline for a small Node.js API — test, containerize, and deploy to staging automatically, then to production behind a manual approval gate — using every technique from the course.

The App

A small Node.js/Express API, nothing fancy — just enough to give the pipeline something real to build, test, and deploy.

⚙️ The Full Workflow

```yaml
name: CI/CD

on: [push, pull_request]

jobs:
  test:
    strategy:
      matrix:
        node-version: [18, 20, 22]  # Ch.4 — matrix build
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4                    # Ch.3 — always first
      - uses: actions/setup-node@v4
        with:
          node-version: ${{ matrix.node-version }}
          cache: 'npm'                               # Ch.4 — lockfile-hashed cache
      - run: npm ci
      - run: npm test

  build-and-push:
    needs: test                                      # Ch.2/Ch.6 — no image without passing tests
    if: github.ref == 'refs/heads/main'              # Ch.2 — only push-to-main deploys
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - run: docker build -t ghcr.io/myorg/api:${{ github.sha }} .  # Ch.6 — SHA tag
      - run: echo "${{ secrets.REGISTRY_TOKEN }}" | docker login ghcr.io -u ${{ github.actor }} --password-stdin
# Ch.5
      - run: docker push ghcr.io/myorg/api:${{ github.sha }}

  deploy-staging:
    needs: build-and-push
    runs-on: ubuntu-latest
    environment: staging                             # Ch.8 — no reviewers, deploys automatically
    steps:
      - run: ./deploy.sh staging ${{ github.sha }}
      - run: curl -f https://staging.example.com/health    # Ch.7 — health check

  deploy-production:
    needs: deploy-staging
    runs-on: ubuntu-latest
    environment: production                          # Ch.8 — required reviewer configured
    steps:
```

```
    - run: ./deploy.sh production ${{ github.sha }}      # Ch.6 — the SAME image, never rebuilt
    - run: curl -f https://example.com/health || ./rollback.sh ${{ github.sha }}  # Ch.7
```

Walking Through a Real Run

1. A developer pushes to `main` — the `test` job runs across three Node versions in parallel
2. Once all three pass, `build-and-push` builds and pushes one SHA-tagged image
3. `deploy-staging` deploys that image automatically, then checks its health endpoint
4. The team reviews staging; a designated reviewer approves the `production` environment gate
5. `deploy-production` deploys the exact same image — never rebuilt — checking health again, with rollback ready if it fails

What's Simplified Here

No database migrations, no canary/blue-green rollout mechanics (Chapter 7 covered the concepts; a real production app would implement one), no monitoring/alerting integration.

What This Capstone Focuses On

The structural correctness of the pipeline itself — every job dependency, every gate, every safety mechanism from Chapters 2–8, wired together correctly.

Chapter 1's Manual Process vs This Pipeline

Manual (Chapter 1)

Remember to run tests, manually build, manually copy files, manually restart — every step a chance to forget something.

This Capstone's Pipeline

Every step automated and gated appropriately — nothing depends on a human remembering anything except the one deliberate approval this app's stakes actually warrant.

💻 Coding Challenges

Challenge 1: Add a Rollback Job for Staging

The `deploy-staging` job currently checks health but doesn't roll back on failure. Add a rollback step to it, mirroring `deploy-production`'s pattern.

Goal: Practice extending the capstone's own pipeline with a missing safety mechanism.

→ Solution

Challenge 2: Trace What Happens on a Pull Request

Trace this capstone's workflow for a pull request (not a push to main) — which jobs run, which don't, and why, citing the specific YAML condition responsible.

Goal: Practice tracing trigger-to-stage behavior through a real, complete workflow file.

→ Solution

Challenge 3: Justify Every Job Dependency

For each of the four `needs:` relationships in this capstone's workflow, state which earlier chapter's principle it enforces and what would go wrong if that dependency were removed.

Goal: Practice holding the entire course's material together as one coherent system.

→ Solution

⚠️ Gotcha: Testing Pipeline Changes Against Production Itself

The pipeline defined in this capstone is itself code — and it deserves the same "test before it affects everyone" caution as the application it deploys. A change to the workflow file (a new job, a modified deploy script) that's merged straight to `main` and immediately exercised against the real staging/production environments risks the exact kind of incident this whole course exists to prevent, just one level up. Test pipeline changes the same way application changes are tested: open a PR first (exercising only the `test` job, per Chapter 2's trigger mapping), and consider a manual `workflow_dispatch` run against a scratch environment before trusting a modified pipeline with real staging or production traffic.

🎓 Course Complete

This closes out **CI/CD Pipelines** — from why automation matters, through pipeline anatomy, GitHub Actions fundamentals, building and testing, secrets, Docker, deployment strategies, environments and approval gates, and finally this capstone assembling a complete, real pipeline end to end.