16

CI/CD Pipelines

From Commit to Confidence

THE MODERN QA ENGINEER SKILLSET

CI/CD Pipelines: From Commit to Confidence

Building, Testing, and Deploying with Automation

Yuri Syuganov

Modern QA Course

https://modernQAcourse.com

CI/CD Pipelines: From Commit to Confidence

Building, Testing, and Deploying with Automation

Book 16 of 26 in the series *The Modern QA Engineer Skillset*

First Edition · Version 1.0.0 · 2026-07-11

Set in Charter, IBM Plex Sans, and IBM Plex Mono. Typeset with Pandoc and XeLaTeX.

Modern QA Course · https://modernQAcourse.com · support@modernQAcourse.com

Contents

Introduction 1
Part I: Foundations 5
1. The CI/CD Landscape 7
About This Sample 17
About the Author 19
Also in This Series 21
Stuck on a Real Problem? 23

Introduction

Every line of code you write is a promise. A promise that the feature works, that nothing else broke, and that users can rely on the software you ship. But promises mean nothing without proof. That proof lives in your CI/CD pipeline – the automated gauntlet that every commit must survive before it reaches a single user.

This book teaches you to build, optimize, and master CI/CD pipelines from the ground up. It is not a surface-level tour of buttons and menus. It is a deep, practical guide that takes you from understanding why pipelines exist to building production-grade systems that give your team genuine confidence in every release.

A test suite that only runs on your laptop does not exist. The pipeline is where tests prove their value – blocking bad code from reaching users, providing fast feedback to developers, and producing artifacts that help diagnose failures. Engineers who can configure, optimize, and troubleshoot pipelines are dramatically more effective than those who treat CI/CD as "someone else's job."

By the end of this book, you will have built a complete CI/CD pipeline from scratch, optimized it for speed, and configured quality gates that make your main branch a fortress.

Who This Book Is For

- **Software engineers** who want to understand the full lifecycle of code from commit to production
- **QA engineers** who want to own pipeline knowledge and become indispensable to their teams

- **DevOps practitioners** looking for a structured approach to pipeline design and optimization
- **Team leads and engineering managers** who want to establish CI/CD best practices across their organization
- **Students and career-changers** entering software engineering who need to understand modern delivery practices

You do not need to be a CI/CD expert to start. You do need curiosity and willingness to get your hands dirty with YAML, terminal commands, and configuration files.

Prerequisites

Before diving in, make sure you are comfortable with the following:

- **Git fundamentals**: Branches, commits, pull requests, and merging. You should be able to create a branch, make changes, push, and open a pull request without hesitation.
- **Command-line basics**: Navigating directories, running scripts, reading terminal output. You will spend significant time reading pipeline logs.
- **A programming language**: The examples use JavaScript/TypeScript and Node.js, but the concepts apply to any language. If you use Python, Go, Java, or anything else, you will be able to translate the patterns.
- **Basic YAML syntax**: Indentation matters. Keys and values. Lists and maps. If you have edited a configuration file before, you are ready.
- **A GitHub account**: Most hands-on exercises use GitHub Actions. A free account provides more than enough pipeline minutes to complete every exercise in this book.

How to Use This Book

Start at Chapter 1 and work forward. Each chapter builds on the previous one. The book is structured in three parts:

Part I – Foundations (Chapters 1-3): Understand the CI/CD landscape, master GitHub Actions in depth, and survey the other major platforms.

Part II – Test Integration (Chapters 4-5): Learn to structure your pipeline around the testing pyramid and manage the artifacts that pipelines produce.

Part III – Optimization and Deployment (Chapters 6-9): Make pipelines fast, deploy with confidence using battle-tested strategies, enforce quality gates, and tie everything together in a capstone project.

Every chapter includes:

- Complete, production-ready YAML configurations (not snippets – full files you can copy and adapt)
- **Pro Tip** callouts with hard-won wisdom from real-world pipelines
- **Common Mistake** callouts that will save you hours of debugging
- Exercises at three levels: Beginner, Intermediate, and Advanced

Pick one platform and learn it deeply. The concepts transfer across all of them.

Part I: Foundations

CHAPTER 1

The CI/CD Landscape

What CI/CD Actually Means

Continuous Integration (CI) is the practice of merging code changes into a shared branch frequently – at least once a day – and validating each merge with automated builds and tests. The key word is "continuous." If you merge once a week after a marathon of local development, you are not doing CI regardless of how many tools you have installed.

Continuous Delivery (CD) extends CI by ensuring that the codebase is always in a deployable state. Every commit that passes the pipeline could be released to production at the push of a button.

Continuous Deployment goes one step further: every commit that passes the pipeline is automatically deployed to production with no human intervention.

Most teams practice CI and Continuous Delivery. Full Continuous Deployment requires an exceptionally mature test suite and monitoring infrastructure. This book will prepare you for all three.

The Pipeline as a Safety Net

Think of your pipeline as a series of increasingly rigorous checkpoints. A commit enters from one end and, if it survives every checkpoint, emerges from the other end as a deployable artifact. Each checkpoint catches a different class of problem:

```
Commit
  |
  v
[Lint & Format]     -- Catches syntax errors, style violations
  |
  v
[Unit Tests]        -- Catches broken logic, regressions
  |
  v
[Integration Tests] -- Catches broken contracts, database issues
  |
  v
[Browser Tests]     -- Catches UI regressions, end-to-end flow failures
  |
  v
[Security Scan]     -- Catches vulnerabilities, dependency issues
  |
  v
[Deploy to Staging] -- Catches environment-specific issues
  |
  v
[Smoke Tests]       -- Catches deployment failures
  |
  v
[Deploy to Production]
```

Each checkpoint is faster and cheaper to fix than the next one. A lint error caught in 30 seconds costs nothing. A broken API discovered after deployment costs an incident.

The Big Four Platforms

As of 2026, most teams use one of four CI/CD platforms. Each has its own configuration format and hosting model, but they all share the same core concepts: triggers, jobs, steps, environment variables, secrets, and artifacts.

Platform	Config Format	Hosted/Self-Hosted	Strengths	Limitations
GitHub Actions	YAML (`.github/workflows/`)	Hosted (+ self-hosted runners)	Native GitHub integration, marketplace of actions, generous free tier	Debugging workflows can be slow (push-and-pray)
GitLab CI	YAML (`.gitlab-ci.yml`)	Both	Built-in container registry, review apps, strong DevSecOps features	YAML complexity grows quickly
Jenkins	Groovy (`Jenkinsfile`)	Self-hosted	Maximum flexibility, massive plugin ecosystem	Maintenance burden, UI feels dated
CircleCI	YAML (`.circleci/config.yml`)	Hosted	Fast execution, excellent caching, orbs for reuse	Cost scales with parallelism

Shared Concepts Across All Platforms

Regardless of which platform you use, the following concepts work the same way. Understanding these makes switching platforms straightforward.

Triggers

Triggers define when a pipeline runs. Common triggers include:

- **Push to branch**: Run on every code push (most common for CI)
- **Pull request / merge request**: Run when a PR is opened or updated
- **Schedule (cron)**: Run on a timer for nightly regressions or data freshness checks
- **Manual dispatch**: Allow humans to trigger pipelines with parameters (useful for deploying to specific environments)
- **Tag creation**: Run when a new release tag is pushed

Jobs and Steps

A **job** is a unit of work that runs on a single machine (runner). Jobs can run in parallel or sequentially depending on dependencies. A **step** is a single command or action within a job.

```
Pipeline
  +-- Job A (runs first)
  |     +-- Step 1: Checkout code
  |     +-- Step 2: Install dependencies
  |     +-- Step 3: Run unit tests
  |
  +-- Job B (depends on Job A)
        +-- Step 1: Checkout code
        +-- Step 2: Run integration tests
        +-- Step 3: Upload test reports
```

Environment Variables and Secrets

Environment variables configure behavior without hardcoding values. Secrets are encrypted environment variables for sensitive data like API keys, passwords, and tokens.

Rules for secrets:

- Never commit secrets to version control
- Use the platform's secret store (GitHub Secrets, GitLab CI Variables, Jenkins Credentials)
- Rotate secrets regularly
- Scope secrets to the minimum required access (repository-level, not organization-level, when possible)

Runners

A runner is the machine where your pipeline executes. Hosted runners are provided by the platform (convenient but with limited customization). Self-hosted runners are machines you manage (more control, better for specialized hardware or internal network access).

When to use self-hosted runners:

- Tests need access to internal services behind a firewall
- You need specialized hardware (GPU, mobile devices)
- Pipeline volume makes hosted runners expensive
- Compliance requirements mandate that code never leaves your network

Choosing a Platform

For most teams, the choice is simple: **use what integrates with your source control**. If your code is on GitHub, use GitHub Actions. If it is on GitLab, use GitLab CI. The friction of using a separate CI platform (e.g., Jenkins with GitHub) rarely justifies the benefits.

Choose Jenkins when:

- You need complex, highly customized pipelines with plugins for niche tools
- You have a dedicated DevOps team to maintain the Jenkins infrastructure
- Your organization already has a mature Jenkins setup

Choose CircleCI when:

- You need the absolute fastest execution times
- Your team uses multiple source control platforms
- You want orbs (reusable pipeline packages) to reduce boilerplate

> **Pro Tip**: Do not agonize over the platform choice. The concepts in this book – triggers, caching, parallelization, quality gates – work on every platform. Pick one and go deep. You can switch later; the knowledge transfers.

> **Common Mistake**: Choosing a platform based on feature lists rather than integration with your existing tools. A platform that does not integrate tightly with your source control adds friction to every pull request, and that friction compounds into wasted hours every week.

Exercises

Beginner

1. Create a free GitHub account if you do not already have one. Create a new repository with a simple README file. Navigate to the Actions tab and observe the suggested starter workflows.
2. Write down the five trigger types covered in this chapter and give a real-world example of when you would use each one.

Intermediate

3. Research the pricing model for GitHub Actions, GitLab CI, and CircleCI. Calculate how many pipeline minutes per month your team would need based on your current commit frequency and average pipeline duration.
4. Set up a self-hosted runner on your local machine for a GitHub repository. Run a simple workflow on it and observe the differences from a hosted runner.

Advanced

5. Design a decision matrix for choosing a CI/CD platform for a new project. Include criteria such as: source control integration, cost at scale, self-hosted runner support, marketplace/plugin ecosystem, secret management, and compliance features. Score each platform against your criteria.

Career Translation

Resume phrasing

- Evaluated and selected CI/CD platforms (GitHub Actions, GitLab CI, Jenkins, CircleCI) based on integration requirements, cost modeling, and team capabilities, reducing pipeline adoption time by 40%.
- Defined CI/CD strategy for a cross-functional engineering team, establishing trigger policies, secret management standards, and runner provisioning guidelines.
- Designed pipeline architectures with staged quality checkpoints (lint, unit, integration, E2E, security), cutting escaped defects to production by over 30%.

Cover letter framing

I bring a systems-level understanding of CI/CD that goes beyond tool configuration. I evaluate platforms against real constraints – source control integration, cost at scale, compliance requirements – and design pipeline architectures that give teams fast feedback without sacrificing thoroughness. My focus is on making the pipeline a trusted safety net so engineers ship with confidence rather than anxiety.

Interview framing

"I approach CI/CD platform selection by first mapping the team's constraints: where does the code live, what's the compliance posture, and what does the team already know. I optimize for integration tightness over feature lists – a platform that adds friction to every PR compounds into lost hours fast. Trade-offs include hosted convenience versus self-hosted control, and I frame those as cost-of-ownership decisions rather than feature comparisons."

What not to say

- "I've used Jenkins" – without explaining what you configured, why, or what trade-offs you managed. Tool familiarity is not the same as pipeline design competence.
- "CI/CD is the DevOps team's job" – signals you treat quality as someone else's responsibility rather than a shared engineering concern.
- "We just run all our tests on every push" – reveals a lack of understanding of pipeline staging, feedback speed, and cost efficiency.
- "I copy YAML from Stack Overflow" – suggests you do not understand the configuration you are shipping, which leads to fragile pipelines nobody can debug.

Interview Depth Check

Question 1

Prompt: Your team is migrating from a monolith on Jenkins to microservices on GitHub. The CTO wants everything moved in one sprint. How do you push back, and what migration plan do you propose?

What a strong answer should cover:

- Risk of a big-bang migration: broken pipelines block all teams simultaneously
- Phased approach: migrate one service at a time, starting with the least critical
- Parallel running period where both systems coexist
- Secret migration strategy and audit trail
- Team training and documentation before cutover

Example answer:

- "I would propose a phased migration starting with a non-critical service to validate our GitHub Actions patterns. During the first phase, both Jenkins and GitHub Actions run in parallel so we have a fallback. I would migrate secrets through the platform's encrypted stores with an audit log, never through code. Each migrated service gets a one-week soak period before we decommission its Jenkins job. This approach limits blast radius and gives the team time to build confidence with the new platform."

Question 2

Prompt: You discover that your CI pipeline runs for 35 minutes on every push, and developers have started merging without waiting for results. The test suite itself is sound. What do you do?

What a strong answer should cover:

- Diagnosis: identify which stages are slow (install, test, deploy)
- Pipeline staging: move expensive tests to PR/merge triggers, keep fast tests on push
- Caching: dependencies, browser binaries, Docker layers
- Parallelization: sharding, matrix strategies
- Cultural fix: making the pipeline fast enough that developers trust it again

Example answer:

- "First I would profile each pipeline stage to find the bottleneck. Then I would restructure triggers so only lint and unit tests run on every push – targeting under 3 minutes. Integration and browser tests move to PR events. I would add dependency caching and test sharding to cut wall-clock time further. The goal is to make the fast-feedback loop fast enough that developers naturally wait for it, restoring trust in the pipeline as a safety net rather than a bottleneck."

Question 3

Prompt: A junior engineer asks why you need both CI and CD – why not just deploy every commit directly? How do you explain the difference and the risks?

What a strong answer should cover:

- CI validates code correctness; CD ensures deployability; Continuous Deployment automates the release
- The maturity ladder: CI is table stakes, CD requires a reliable test suite, Continuous Deployment requires exceptional test coverage and monitoring
- Risk without CI: broken code reaches production because nobody validated it
- Risk of premature Continuous Deployment: insufficient test coverage means bugs auto-deploy

Example answer:

- "CI is about proving the code works – every merge triggers automated validation. CD extends that by ensuring the artifact is always deployable, but a human still decides when to release. Continuous Deployment removes that human step entirely. The risk of jumping straight to Continuous Deployment is that your test suite becomes your only safety net – if coverage is thin, bugs auto-deploy to users. I recommend teams earn Continuous Deployment by first building a mature CI practice, then graduating to CD, and only adopting full

Continuous Deployment when their test suite and monitoring give them genuine confidence."

About This Sample

You have just read **Chapter 1 of CI/CD Pipelines: From Commit to Confidence** — the free sample. The complete book continues with every remaining chapter, the interview Q&A, and all of the code.

Get the full book

Sample readers get a discount at
https://leanpub.com/modernqa-16-cicd-pipelines/c/SAMPLE10

Read online first

Chapter one of **every** book in the series is free to read at
https://modernqacourse.com/library

About the Author

Yuri Syuganov is a software quality engineer with more than twenty years of hands-on testing experience, spanning the arc from manual test design through Selenium-era automation to today's AI-augmented, agent-driven frameworks. He is the author of The Modern QA Engineer Skillset, a 26-book series built on a blunt analysis of what companies actually expect from QA engineers, and the creator of modernQAcourse.com, where the course behind the series is free and readers can bring real testing problems to experienced practitioners. He revises the series as the industry changes — because it does.

Also in This Series

The Modern QA Engineer Skillset — a 26-book course in modern software quality assurance.

AI-Augmented QA (Books 1–10)

- **Book 1:** Agent Skills for Browser Automation: The Complete Guide
- **Book 2:** AI-Augmented Test Design: From Prompts to Test Suites
- **Book 3:** Agentic Testing Architectures: Building Autonomous QA Systems
- **Book 4:** API and Contract Testing with AI: Schema to Shield
- **Book 5:** Performance and Chaos Engineering: Breaking Things on Purpose
- **Book 6:** Observability-Driven Testing: Testing in Production
- **Book 7:** Security Testing for AI Applications: The New Attack Surface
- **Book 8:** Infrastructure as Code Testing: Terraform to Kubernetes
- **Book 9:** Mobile and Cross-Platform Testing: 24,000 Devices, One Strategy
- **Book 10:** Visual and Accessibility Testing: Pixels and People

Technical Foundations (Books 11–19)

- **Book 11:** Manual Testing Fundamentals: The Bedrock
- **Book 12:** Programming for QA: Python, JavaScript, and TypeScript from Zero to Fluent
- **Book 13:** Browser Test Automation: Playwright, Patterns, and the Death of Fragile Tests

- **Book 14:** API Testing Fundamentals: Postman to Pytest
- **Book 15:** SQL and Database Testing: Query Your Way to Quality
- **Book 16:** CI/CD Pipelines: From Commit to Confidence
- **Book 17:** Git and Version Control: Beyond Push and Pull
- **Book 18:** Test Management Tools: Making Quality Visible
- **Book 19:** Linux and Command Line: Your Production Survival Kit

Professional Skills (Books 20–26)

- **Book 20:** Agile and Scrum for QA: Quality in Every Sprint
- **Book 21:** Communication and Stakeholder Management for QA: Influence Without Authority
- **Book 22:** Test Strategy and Quality Metrics: Measuring What Matters
- **Book 23:** QA Leadership and Mentoring: Multiplying Your Impact
- **Book 24:** Technical Writing for QA: Documents That Drive Action
- **Book 25:** Interview Preparation and Career Strategy: Land the Job
- **Book 26:** Testing Like a Senior: Patterns & Anti-Patterns Across Software QA Career Levels

Visit https://modernQAcourse.com for the full course catalog, updates, and companion resources.

Stuck on a Real Problem?

The full course behind this series is free to read at https://modernQAcourse.com — and it stays free.

When a chapter meets the real world and something breaks, post it on the **Help Board** at https://modernQAcourse.com. Asking is free, and experienced practitioners answer right in the thread. Need hands-on help? Book a 1:1 screen-share session with a QA expert.