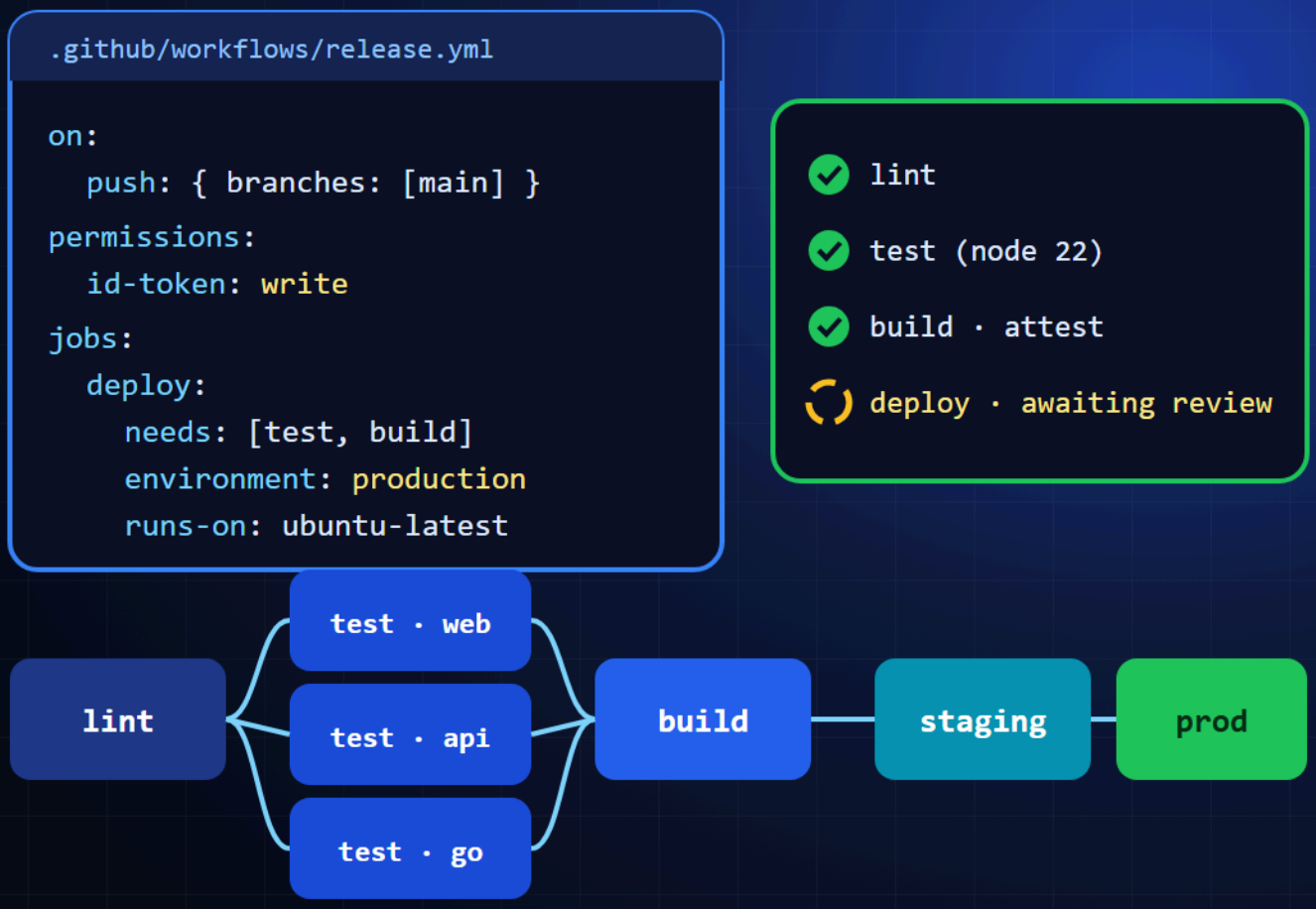


push · pull_request · merge_group · workflow_call · OIDC

THE PRACTICAL GUIDE

CI/CD with GitHub Actions

From first workflow to production pipelines

Build, test, secure, release, and deploy — Node.js, Python, Go, containers, Kubernetes, and Terraform

WORKFLOWS · CI · REUSE · SECURITY · DEPLOYMENT

THENEXBOOKS

CI/CD with GitHub Actions

From First Workflow to Production Pipelines
Build, Test, Secure, Release, and Deploy with Confidence

First Edition

THENEXBOOKS

CI/CD with GitHub Actions
From First Workflow to Production Pipelines

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. GitHub Actions, its runners, marketplace actions, pricing, and usage limits change frequently; always consult the official GitHub documentation, changelog, and billing pages for current details.

GitHub and GitHub Actions are trademarks of GitHub, Inc. This book is not affiliated with, sponsored by, or endorsed by GitHub, Inc. or Microsoft Corporation. Docker, Kubernetes, Terraform, AWS, Azure, Google Cloud, Node.js, Python, Go, and other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement.

Harvestly, the running example in this book, is fictional, as are all farms, customers, organizations, and people mentioned in its examples. Any resemblance to real companies or products is coincidental.

Preface

Every team that ships software eventually discovers the same truth: the path from a commit to production is itself a product. When it is slow, developers wait and batch up risky changes. When it is flaky, people stop trusting red builds. When it is insecure, a single leaked token or poisoned dependency can compromise everything downstream. And when it lives in one person's head, releases stop whenever that person is on vacation.

GitHub Actions puts that path right next to the code. Workflows are YAML files in the repository, reviewed in pull requests like any other change, triggered by the events your team already generates — pushes, pull requests, releases, schedules, and more. Getting a first workflow to run takes minutes. Building pipelines that stay fast, trustworthy, secure, and maintainable across dozens of repositories and teams takes real engineering, and that is what this book is about.

We start from the first workflow and the core syntax, build a complete continuous integration pipeline for a multi-language monorepo, and then go deep on reuse, runners, security, releases, deployments, and operations — always explaining *why* a pattern works, where it breaks, and what goes wrong in production.

Who This Book Is For

- **Developers** who want reliable automated builds, tests, and releases for their projects.
- **DevOps and platform engineers** who build shared pipelines, runners, and deployment tooling for many teams.
- **Teams migrating** from other CI systems who want to learn how GitHub Actions thinks.
- **Tech leads and security engineers** responsible for pipeline governance, least privilege, and supply-chain security.

What You Will Learn

- Workflows, events, jobs, steps, runners, expressions, contexts, and conditionals — precisely and in depth.
- Complete CI pipelines for Node.js, Python, and Go, with matrices, service containers, caching, and artifacts.
- Pull request workflows with required checks, merge queues, and path filtering for monorepos.
- Reusable workflows, composite actions, and JavaScript and container actions of your own.
- GitHub-hosted, larger, and self-hosted runners, including runners on Kubernetes, and organization-wide governance.
- Least-privilege tokens, secrets and environments, OpenID Connect to cloud providers, and workflow hardening.
- Supply-chain security with Dependabot, code scanning, SBOMs, and artifact attestations.

- Container images and packages, release automation, cloud and Kubernetes deployments, Terraform pipelines, and progressive delivery.
- Debugging, observability, performance, and cost optimization for pipelines.

How This Book Is Organized

Part I — GitHub Actions Foundations (Chapters 1–5) explains the platform, the workflow language, events, and marketplace actions. **Part II — Continuous Integration** (Chapters 6–10) builds a complete CI pipeline for a multi-language monorepo. **Part III — Reuse and Scale** (Chapters 11–15) turns one-off workflows into shared building blocks and covers runners and organization governance. **Part IV — Security** (Chapters 16–20) locks down tokens, secrets, cloud access, workflow code, and the software supply chain. **Part V — Continuous Delivery and Deployment** (Chapters 21–25) publishes artifacts, automates releases, and deploys safely. **Part VI — Operating Pipelines and Case Study** (Chapters 26–28) covers debugging, performance, cost, and a complete end-to-end pipeline. The appendices provide a cheat sheet, workflow templates and a syntax reference, a production pipeline checklist, and a glossary.

Each chapter ends with a **Key Takeaways** box for quick review.

The Running Example: Harvestly

Throughout the book we build the pipelines of **Harvestly**, a fictional farm-to-door grocery subscription platform. Its main repository, `harvestly/platform`, is a monorepo with a TypeScript and Next.js storefront, a Python API, a Go pricing engine, a shared UI package, Terraform infrastructure, and Kubernetes deployment manifests. That mix lets us show realistic pipelines for several languages at once, along with the problems that appear only at scale: slow builds, flaky tests, shared workflows across teams, cloud credentials, container images, and releases that must never take the storefront down on a Saturday morning.

Versions and Conventions

Examples use current major versions of official and widely used actions at the time of writing, such as `actions/checkout@v4`. Actions release new major versions over time, so check each action's releases before copying a version into your own workflows — and in Chapter 19 we explain why production workflows should pin actions to full commit SHAs. GitHub's prices, included minutes, and usage limits change; where they matter, we point you to the official documentation rather than quoting numbers that may be out of date. Features in public preview are labeled as such.

> **TIP**
>
> A recommended practice or a technique that makes your work easier.

> **NOTE**
> Background information, version details, or a useful aside.

> **WARNING**
> A common mistake or a behavior that can cause broken builds, leaked secrets, or failed deployments.

Code appears in `monospace`. Long lines are wrapped to fit the page, and workflow files live under `.github/workflows/` unless stated otherwise.

Contents

Preface

PART I — GITHUB ACTIONS FOUNDATIONS

1 Why CI/CD, and Why GitHub Actions

2 Your First Workflow: Events, Jobs, Steps, and Runners

3 Workflow Syntax in Depth: Expressions, Contexts, Variables, and Conditionals

4 Events and Triggers: push, pull_request, schedule, workflow_dispatch, and More

5 Using Marketplace Actions: Versions, Inputs, Outputs, and Trust

PART II — CONTINUOUS INTEGRATION

6 Building a CI Pipeline: Lint, Test, and Build

7 Matrix Builds and Multi-Language Pipelines (Node.js, Python, Go)

8 Service Containers and Integration Testing

9 Caching, Artifacts, and Faster Pipelines

10 Pull Request Workflows: Required Checks, Merge Queues, and Monorepo Path Filtering

PART III — REUSE AND SCALE

11 Reusable Workflows

12 Writing Composite Actions

13 Writing JavaScript and Docker Container Actions

14 Runners: GitHub-Hosted, Larger, Self-Hosted, and Actions Runner Controller

15 Actions at Organization Scale: Templates, Policies, and Governance

PART IV — SECURITY

16 GITHUB_TOKEN, Permissions, and Least Privilege

17 Secrets, Variables, and Environments

18 OpenID Connect: Keyless Authentication to Cloud Providers

19 Hardening Workflows: Script Injection, pull_request_target, and Pinning Actions

20 Supply Chain Security: Dependabot, Code Scanning, SBOMs, and Artifact Attestations

PART V — CONTINUOUS DELIVERY AND DEPLOYMENT

21 Building and Publishing Container Images and Packages

22 Release Automation: Versioning, Changelogs, and GitHub Releases

23 Deploying to Cloud Platforms and Kubernetes

24 Infrastructure as Code Pipelines with Terraform

25 Progressive Delivery: Approvals, Canary, Blue-Green, and Rollbacks

PART VI — OPERATING PIPELINES AND CASE STUDY

26 Debugging, Troubleshooting, and Observability for Workflows

27 Pipeline Performance and Cost Optimization

28 Case Study: Harvestly's Pipeline from First Commit to Production

APPENDICES

A GitHub Actions Cheat Sheet

B Workflow Templates and Syntax Reference

C Production Pipeline Checklist

D Glossary

PART I

GitHub Actions Foundations

What CI/CD is for and how GitHub Actions implements it: workflows, events, jobs, steps, and runners; the expression language and contexts; every important trigger; and how to choose and use marketplace actions safely.

1 · Why CI/CD and GitHub Actions
2 · Your First Workflow
3 · Workflow Syntax in Depth
4 · Events and Triggers
5 · Using Marketplace Actions

CHAPTER 1 · PART I — GITHUB ACTIONS FOUNDATIONS

Why CI/CD, and Why GitHub Actions

Continuous integration and delivery are less about tools than about shortening the distance between a change and the confidence that it works. This chapter explains what CI/CD really means, what it costs, why GitHub Actions has become a natural home for it, and introduces Harvestly — the grocery subscription platform whose pipelines we build throughout the book.

A Friday Afternoon at Harvestly

Picture a small engineering team at Harvestly, a farm-to-door grocery subscription service. Customers subscribe to weekly produce boxes, farms such as Greenacre Farm and Millbrook Orchards list what they are harvesting, and a pricing engine decides what each box costs this week. The code lives in a single GitHub repository with a storefront, an API, and the pricing service side by side.

It is Friday afternoon. A developer merges a small change to the pricing engine: a rounding fix for boxes that mix weighed and counted items. The change looks obviously correct, the unit tests pass on the developer's laptop, and a teammate approved the pull request after a quick read. Someone then SSHes into a build server, pulls the main branch, runs a build script that was last edited eight months ago, copies a binary to production, and restarts a process.

On Saturday morning, support tickets arrive. Some boxes are priced at zero. The rounding fix was fine; the problem was that the build server still had an older version of a dependency cached, and the binary that shipped was not built from exactly the code that was reviewed. Nobody can say for certain which commit is running in production, the build log scrolled off a terminal that has since been closed, and rolling back means repeating the same manual ritual in reverse — carefully, on a weekend, under pressure.

None of the people involved did anything unusual. Every step was reasonable in isolation. What was missing is a **system**: an automated, repeatable path from a commit to a running service, in which every change is built the same way, tested the same way, recorded, and deployable or reversible with a single well-understood action. That system is what CI/CD provides, and building it well is what this book is about.

What CI/CD Actually Means

The acronym bundles three related practices. They are often blurred together, but separating them makes it much easier to decide what your team needs next.

Continuous integration

Continuous integration (CI) is the practice of merging every developer's work into a shared mainline frequently — at least daily — and verifying each merge automatically. The verification is the part people usually picture: a server checks out the code, installs dependencies, compiles, runs linters and tests, and reports a green check or a red cross. But the *integration* part matters just as much. Long-lived branches that diverge for weeks produce painful merges and surprising interactions; frequent small merges keep each integration boring.

CI therefore has two halves. The cultural half is small batches and a mainline that is always expected to work. The technical half is an automated build that is fast and trustworthy enough that people actually wait for it and believe it. When either half is missing, CI degrades into "a server that runs tests sometimes."

Continuous delivery

Continuous delivery extends CI so that every change that passes the pipeline produces a *releasable* artifact — a container image, a package, a bundle of static files — and the path to production is automated up to, but not necessarily including, the final push of the button. In a continuous delivery setup, releasing is a business decision rather than an engineering project. Someone approves a deployment to production, and the same automation that deployed to staging an hour ago does the rest.

Continuous deployment

Continuous deployment removes the manual approval: every change that passes all automated checks goes to production automatically. This demands a lot from the pipeline — strong test coverage, safe database migrations, progressive rollout with automatic rollback, and good monitoring — because there is no human gate to catch what the tests miss. Many teams practice continuous deployment for some services (a static storefront, an internal tool) and continuous delivery with an approval step for others (a payments path, a pricing engine).

Practice	Automated up to	Human step	Main prerequisite
Continuous integration	Build and test of each change	Merge decision	Fast, reliable test suite
Continuous delivery	A releasable artifact, deployed to pre-production	Approve production release	Repeatable, scripted deploys
Continuous deployment	Production	None in the normal path	Safe rollout, monitoring, rollback

Throughout this book, "CI/CD" means the whole spectrum, and we build Harvestly's pipelines to support both styles: continuous deployment to the `dev` environment, and continuous delivery with approvals for `staging` and `production`.

What a Pipeline Buys You

It is tempting to justify CI/CD with dramatic numbers. We will not do that — any figure would depend entirely on your codebase, team, and starting point. Instead, it is worth understanding the *mechanisms* by which a good pipeline helps, because those mechanisms also tell you how to design one.

Short feedback loops

A bug found minutes after it was written is cheap: whoever wrote it still has the context in their head, the change is small, and nothing else has been built on top of it. The same bug found in production weeks later is expensive in every way. A CI pipeline that runs on every push and every pull request moves detection as early as possible. That is also why pipeline *speed* matters so much: a check that takes an hour gets ignored, batched, or bypassed, and the feedback loop stretches back out. We spend Chapters 9 and 27 on making pipelines fast.

Reproducibility

The Friday incident happened because the artifact that shipped was built in an environment nobody controlled. A pipeline builds on a fresh, defined machine every time, from a specific commit, with dependencies resolved from lock files. When something goes wrong, you can rebuild exactly the same thing, or build the previous commit and roll back with confidence.

An audit trail

Every workflow run records who or what triggered it, which commit it used, what every step printed, and whether it succeeded. Deployments linked to environments show what is running where. For teams with compliance obligations, this record — combined with branch protection and required reviews — is often exactly the evidence auditors ask for. For everyone else, it answers the most common incident question: "what changed?"

Enforced standards without nagging

Code review is a poor place to argue about formatting, import order, or whether a test was added. A pipeline can enforce formatting, linting, license checks, and test requirements mechanically, so humans can review the design and the logic. The pipeline becomes the neutral referee that nobody takes personally.

Safer, smaller, more frequent releases

When releasing is scripted and cheap, teams release more often, and each release contains fewer changes. Smaller releases are easier to reason about, easier to roll back, and easier to bisect when something breaks. This is the quiet compounding benefit of continuous delivery: it changes the risk profile of every individual deployment.

What CI/CD Costs

A pipeline is software, and like all software it has costs that are easy to underestimate. Being honest about them early helps you avoid the most common failure modes.

- **Maintenance.** Workflow files, build scripts, and the actions they depend on need updating. Runner images change, tools release new major versions, and deprecated features stop working. Budget time for pipeline upkeep just as you would for any dependency.
- **Compute.** Every run consumes runner time. On GitHub-hosted runners, minutes for private repositories are billed beyond an included allowance; self-hosted runners cost whatever your infrastructure costs. Wasteful pipelines — running everything on every change, never caching, never cancelling superseded runs — cost real money.
- **Flakiness.** A test that fails randomly one run in twenty teaches everyone to click "re-run" without reading the failure. Once people stop trusting red builds, the pipeline has lost most of its value. Treat flaky tests as bugs with an owner.
- **Security surface.** A pipeline holds credentials that can publish packages and deploy to production. It runs code from pull requests. It pulls third-party actions. A careless pipeline is one of the most attractive targets in your organization, which is why Part IV of this book is devoted entirely to security.
- **Waiting.** Required checks and approvals slow individual merges down. That is often the point, but a pipeline that blocks every change for a long time pushes developers toward bigger batches — the opposite of what CI intends.

> **WARNING**
> A pipeline that is red most of the time is worse than no pipeline at all, because it trains people to ignore failures. If you inherit a permanently failing build, the first job is to get it green — even by temporarily quarantining broken tests with a tracked issue — and only then add new checks.

Where GitHub Actions Fits

GitHub Actions is GitHub's built-in automation platform. It is often described as "GitHub's CI service," and CI is certainly its most common use, but the more accurate description is broader: GitHub Actions runs **workflows** in response to **events** that happen in or around a repository. A push is an event. So is opening a pull request, publishing a release, adding a label to an issue, a scheduled time arriving, a merge queue asking for a check, or a person clicking a "Run workflow" button. Anything you can script, you can run in response to any of those events.

That event-driven design is the first reason Actions is a good fit for CI/CD on GitHub. The same platform that tests pull requests can also triage issues, publish packages when a release is created, rotate stale branches on a schedule, and deploy when a tag is pushed. You do not need a separate integration, webhook receiver, or bot account for each of those jobs.

Reasons teams choose it

- **It lives next to the code.** Workflows are YAML files in `.github/workflows/`, versioned with the code they build, reviewed in the same pull requests, and branched along with everything else. A change that needs a new build step ships with that build step.
- **No server to run.** GitHub-hosted runners are fresh virtual machines provisioned for each job with Linux, Windows, and macOS images that already contain common toolchains. You can start without operating any infrastructure, and move to larger or self-hosted runners later (Chapter 14).
- **Deep platform integration.** Check results appear on pull requests, required status checks feed branch protection and rulesets, environments tie deployments to approvals, and the automatically issued `GITHUB_TOKEN` lets a workflow talk to the GitHub API without you managing a personal token.
- **A large ecosystem of reusable actions.** Setting up a language toolchain, logging in to a registry, or building a container image is usually one step using an action someone has already written. That ecosystem is powerful and also a supply-chain risk; Chapter 5 and Chapter 19 teach you to use it safely.
- **Reuse at scale.** Composite actions, JavaScript actions, reusable workflows, and organization templates let a platform team offer a paved road that dozens of repositories share (Part III).
- **Keyless cloud access.** OpenID Connect lets workflows obtain short-lived cloud credentials without storing long-lived keys as secrets — covered in depth in Chapter 18.

The mental model

Almost everything in GitHub Actions fits into one hierarchy. Learn it now and the rest of the book becomes a matter of details.

```
event (push, pull_request, schedule, workflow_dispatch, ...)
  |
  v
workflow  (.github/workflows/pricing-ci.yml)
  |
  +-- job: lint ----------- runs-on: ubuntu-latest  (one fresh runner)
  |     +-- step: checkout            (uses: an action)
  |     +-- step: run linter          (run: a shell command)
  |
  +-- job: test ----------- runs-on: ubuntu-latest  (another runner)
  |     +-- step: checkout
  |     +-- step: go test ./...
  |
  +-- job: build ---------- needs: [lint, test]
        +-- step: checkout
        +-- step: go build
```

- An **event** triggers one or more **workflows** whose `on:` section matches it.

- A **workflow** contains one or more **jobs**. By default, jobs run in parallel; `needs:` creates dependencies between them.
- Each **job** runs on its own **runner** — a machine with the runner application installed. Jobs do not share a filesystem; passing data between them requires outputs or artifacts.
- A job is a sequence of **steps** executed in order on that runner. A step either runs shell commands (`run:`) or invokes an **action** (`uses:`), which is a reusable unit of automation.

Chapter 2 walks through each of these layers by building a real workflow. For now, the key insight is that the *job* is the unit of isolation and scheduling, and the *step* is the unit of work.

A First Look at a Workflow

Before we dive into syntax, here is a complete, working workflow for Harvestly's pricing engine, so you can see the shape of the thing. Do not worry about every line yet; Chapters 2 and 3 explain all of it.

`.github/workflows/pricing-ci.yml`

```
name: Pricing CI

on:
  push:
    branches: [main]
    paths:
      - "services/pricing/**"
  pull_request:
    paths:
      - "services/pricing/**"

permissions:
  contents: read

jobs:
  test:
    runs-on: ubuntu-latest
    defaults:
      run:
        working-directory: services/pricing
    steps:
      - name: Check out the repository
        uses: actions/checkout@v4

      - name: Set up Go
        uses: actions/setup-go@v5
        with:
          go-version-file: services/pricing/go.mod

      - name: Verify formatting
        run: test -z "$(gofmt -l .)"

      - name: Run unit tests
        run: go test -race ./...
```

Read it top to bottom and it almost explains itself. The workflow is called "Pricing CI." It runs on pushes to `main` and on pull requests, but only when files under `services/pricing/` change. It restricts the automatic token to read-only access to repository contents. It has one job, `test`, that runs on a fresh Ubuntu machine, checks out the code, installs the Go version declared in `go.mod`, fails if any file is not `gofmt`-formatted, and runs the tests with the race detector.

When this file is committed to the default branch, GitHub starts running it on the matching events. The results appear in the repository's **Actions** tab and, for pull requests, as a check at the bottom of the conversation — a check that can be made mandatory before merging, as we do in Chapter 10.

> **NOTE**
>
> Examples in this book use `actions/checkout@v4`, `actions/setup-go@v5`, and similar major-version tags that were current at the time of writing. Newer major versions may exist by the time you read this — check each action's releases page. Chapter 19 explains why production workflows often pin actions to a full commit SHA instead of a tag.

GitHub Actions Compared With Other CI Systems

GitHub Actions is not the only way to run pipelines, and it is not automatically the right choice for every team. The major alternatives share the same core ideas — pipelines as code, jobs, stages, agents or runners — and differ mostly in where they run, how they integrate with your code host, and how much you operate yourself.

System	Configuration	Hosting model	Typical strength
GitHub Actions	YAML in the repo	Hosted or self-hosted runners	Native GitHub integration, event breadth
GitLab CI/CD	YAML in the repo	Hosted or self-managed	Native GitLab integration
Jenkins	Groovy pipelines	Self-operated	Extreme flexibility, plugins
Azure Pipelines	YAML in the repo	Hosted or self-hosted agents	Azure DevOps integration
CircleCI and others	YAML in the repo	Hosted or self-hosted	Dedicated CI focus

If your code lives on GitHub, Actions removes a whole category of integration work: no webhooks to configure, no separate identity system, no status-reporting plugin. That convenience is the strongest argument for it. The arguments against are usually specific: an existing heavy investment in another system, unusual build hardware you cannot express as a runner, or a regulated environment where the existing tool is already certified. Migrations are possible — GitHub provides tooling to help translate pipelines from several other systems; check the GitHub documentation on migrating to Actions — but a migration is a project and deserves its own plan.

> **TIP**
>
> If you are coming from another CI system, map its concepts before you translate files. A Jenkins "stage" or a GitLab "stage" is roughly a group of Actions jobs linked by `needs:`; an "agent" or "executor" is a runner; a "shared library" or "template" is closest to a reusable workflow or composite action. Translating concepts first produces far cleaner workflows than translating syntax line by line.

Billing, Limits, and the Things That Change

This book deliberately avoids quoting prices, free-minute allowances, storage quotas, timeouts, concurrency limits, or runner hardware specifications. These numbers change, they differ between plans, and a book that states them as fact would become wrong quietly. What stays stable is the model:

- Workflows on GitHub-hosted standard runners are free for public repositories.
- For private repositories, GitHub-hosted runner minutes are billed beyond an allowance included with your plan, and different operating systems and runner sizes are billed at different rates.
- Artifacts, caches, and packages consume storage, which also has plan-dependent allowances.
- Jobs, workflow runs, API calls, and concurrent jobs are subject to usage limits.
- Self-hosted runners do not consume GitHub-hosted minutes, but you pay for and operate the machines.

GitHub documents the current limits in its usage-limits and billing documentation, and you should read those pages for your plan before designing anything that pushes against them. Throughout the book, we design around the *concepts* — cancel redundant runs, cache dependencies, run only what a change affects, set explicit timeouts — which help regardless of the exact numbers. When you want to know whether an optimization worked, measure your own workflow run times before and after rather than trusting anyone's claimed percentage.

Meet Harvestly

Every chapter in this book uses the same running example so that the pipelines grow together rather than appearing as disconnected snippets. Harvestly is fictional, but its shape is deliberately typical: a monorepo with several languages, a shared component library, containerized services, infrastructure as code, and three deployment environments.

Harvestly's code lives in the GitHub organization `harvestly`. The main repository is `harvestly/platform`:

```
harvestly/platform
|-- .github/
|   |-- workflows/          # all workflow files live here
|   |-- CODEOWNERS
|   `-- dependabot.yml
```

```
|-- apps/
|   `-- web/                # storefront: TypeScript + Next.js, Vitest, Playwright
|-- services/
|   |-- api/                # REST API: Python + FastAPI, pytest, PostgreSQL, Redis
|   `-- pricing/            # pricing engine: Go, go test
|-- packages/
|   `-- ui/                 # shared components, published as @harvestly/ui
|-- infra/                  # Terraform for cloud infrastructure
`-- deploy/                 # Kubernetes manifests and Helm chart
```

Two supporting repositories complete the picture. `harvestly/actions` holds the organization's shared composite and JavaScript actions, such as `harvestly/actions/setup-toolchain`, which we build in Part III. `harvestly/workflows` holds reusable workflows and workflow templates that other repositories call.

Container images are published to GitHub Container Registry as `ghcr.io/harvestly/api`, `ghcr.io/harvestly/pricing`, and `ghcr.io/harvestly/web`. The shared UI library is published to GitHub Packages as an npm package. Deployments target three environments — `dev`, `staging`, and `production` — with AWS as the primary cloud example (EKS for Kubernetes, ECR and S3 where relevant) and notes on the Azure and Google Cloud equivalents. Teams referenced in ownership files are `@harvestly/platform-team`, `@harvestly/web-team`, and `@harvestly/sre`.

Why a monorepo with three languages? Because it exercises nearly every interesting feature of GitHub Actions. Path filters decide which services a change affects. Matrix builds test the storefront across Node.js versions. Service containers give the API a real PostgreSQL and Redis during tests. Caching matters because three dependency ecosystems each have their own cache. And deployments must coordinate several images built from one commit. If your own codebase is a single-language repository, every technique still applies — you will just need fewer of them at once.

How This Book Is Organized

The book is divided into six parts that follow the natural growth of a team's automation.

1. **Part I — GitHub Actions Foundations** (Chapters 1–5). The building blocks: workflow anatomy, runners, expressions and contexts, events and triggers, and how to consume Marketplace actions responsibly.
2. **Part II — Continuous Integration** (Chapters 6–10). A real CI pipeline for Harvestly: lint, test, build, matrix builds across languages, service containers for integration tests, caching and artifacts, and pull request workflows with required checks, merge queues, and monorepo path filtering.
3. **Part III — Reuse and Scale** (Chapters 11–15). Reusable workflows, composite, JavaScript, and Docker container actions, runner choices from GitHub-hosted to Actions Runner Controller on Kubernetes, and organization-wide templates and governance.

4. **Part IV — Security** (Chapters 16–20). Least-privilege tokens, secrets and environments, OpenID Connect, hardening against script injection and dangerous triggers, and supply-chain protections such as Dependabot, code scanning, SBOMs, and artifact attestations.
5. **Part V — Continuous Delivery and Deployment** (Chapters 21–25). Publishing container images and packages, release automation, deploying to cloud platforms and Kubernetes, Terraform pipelines, and progressive delivery with approvals, canaries, blue-green deploys, and rollbacks.
6. **Part VI — Operating Pipelines and Case Study** (Chapters 26–28). Debugging and observability, performance and cost, and an end-to-end case study that assembles Harvestly's full pipeline from first commit to production.

The appendices provide a cheat sheet, a syntax reference with workflow templates, a production pipeline checklist, and a glossary. If you are new to Actions, read Part I in order; after that, the parts are designed to be read in sequence but can be dipped into when a specific need arises.

Setting Up to Follow Along

You do not need anything exotic to work through the examples. Here is a practical setup.

1. **A GitHub account and a practice repository.** Create a repository of your own — public repositories are free to run on standard GitHub-hosted runners, which makes experimentation painless. You can mirror the Harvestly layout with placeholder code, or apply the workflows to a project you already have.
2. **Git and an editor with YAML support.** Workflow files are YAML, and indentation errors are the most common beginner mistake. An editor extension for GitHub Actions adds schema validation and autocompletion for workflow keys, which catches many errors before you push.
3. **The GitHub CLI (`gh`).** It lets you list runs, watch them live, re-run failed jobs, and download logs from the terminal. We use it frequently.
4. **Actions enabled.** Actions are enabled by default on new personal repositories. In organizations, administrators can restrict which actions are allowed; if your first workflow does not start, check the repository's **Settings → Actions → General** page, or ask your administrator.

A few `gh` commands you will use constantly:

```
# List recent workflow runs in the current repository
gh run list --limit 10

# Watch a run until it finishes, streaming its status
gh run watch

# Show the logs of only the failed steps of a run
gh run view 1234567890 --log-failed

# Re-run only the failed jobs of a run
gh run rerun 1234567890 --failed
```

```
# Trigger a workflow that has a workflow_dispatch trigger
gh workflow run pricing-ci.yml --ref main
```

You may also come across `nektos/act`, a third-party tool that runs workflows locally in containers. It is handy for quick iteration on simple workflows, but it does not reproduce GitHub-hosted runners exactly, and features that depend on GitHub's infrastructure behave differently or not at all. Treat it as a convenience, not as proof that a workflow works; the real test is always a run on GitHub. Chapter 26 discusses debugging strategies in more depth.

A word on working safely while learning

Experimenting with workflows is low risk as long as you keep one habit from the start: never paste real credentials into workflow files, and never echo secrets in logs to "see if they work." GitHub masks registered secrets in logs, but masking is a safety net, not a design. Start every workflow with a minimal `permissions:` block, as in the pricing example above. Doing this from day one costs nothing and saves you from unlearning a bad habit later; Chapter 16 explains the reasoning in full.

What Good Looks Like

By the end of this book, Harvestly's pipeline will have a clear set of properties, and it is worth stating them up front so you can measure your own pipelines against the same bar:

- Every pull request gets fast, relevant feedback: only the services it touches are built and tested, and superseded runs are cancelled.
- The main branch is protected by required checks and, where traffic warrants it, a merge queue.
- Every artifact deployed anywhere is built once, from a known commit, by the pipeline — never on a laptop — and carries provenance.
- Workflows run with least-privilege tokens, cloud access uses short-lived OIDC credentials, and third-party actions are pinned and reviewed.
- Deployments to `staging` and `production` go through environments with protection rules, and rolling back is a routine, rehearsed action.
- Shared logic lives in reusable workflows and actions, so improving the pipeline once improves it everywhere.

None of this has to happen at once. The most successful pipelines grow incrementally: a single test job, then a lint step, then caching, then a build, then a deployment to one environment. Each increment is small, reviewable, and immediately useful — which is, of course, exactly the philosophy of CI/CD applied to the pipeline itself.