GitHub Actions and CI/CD Course

Flavio Copes

Updated August 3, 2026

Contents

Build a secure GitHub Actions pipeline for tests, artifacts, releases, protected deployments, rollback, and ongoing operations.

What you'll learn: Build and threat-model a pipeline that tests a pull request, creates one artifact, deploys through a protected environment, and rolls back safely.

Workflow foundations

Map events, jobs, steps, runners, contexts, and reproducible commands.

Map events, workflows, jobs, and steps

Understand which repository event starts which jobs and how ordered steps run on a runner.

Before choosing a tool or writing more code, make the requirement concrete. A pull request starts tests, while a push to the default branch can start a later deployment path.

A workflow is event-driven automation; jobs form a dependency graph and steps within one job share its runner workspace. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

A common failure is to copy a large YAML file without drawing the event and job graph. The happy path may still work, which makes the mistake easy to miss. write the trigger and job outcomes in plain language, then encode the smallest matching workflow. Keep the first version small enough that every important input and output remains visible.

This supports the module goal: Create one workflow whose trigger, runner, permissions, commands, and failure behavior are obvious. Capture the request, command, trace, screenshot, test result, or other evidence that proves the result.

Draw the event-to-job graph and annotate which state is shared and which job gets a fresh runner. Change one condition on purpose, predict the result, and compare the prediction with what actually happened.

Choose triggers carefully

Limit branches, paths, activities, schedules, and manual inputs to the events that should spend authority and compute.

Start from the behavior you can observe. Documentation-only changes can skip a heavy build, and deployment runs only from the protected default branch.

Trigger configuration is part of security and cost because it decides which code runs and with what context. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

It is tempting to trigger privileged work on every event and filter it later inside a shell command. That trades a clear decision for an assumption you will eventually have to debug. The stronger move is to narrow events in workflow syntax and test pull request, fork, branch, tag, and manual cases. Make the boundary explicit in code and in the project notes.

Tie the decision back to the larger job: Create one workflow whose trigger, runner, permissions, commands, and failure behavior are obvious. Record enough evidence that another developer can repeat the result without relying on your memory.

Build a trigger table and create one commit for each important included and excluded path. Repeat it once with an invalid or hostile condition and write down the boundary the failure revealed.

Run reproducible steps

Make workflow commands match documented local commands and pin the runtime they require.

The workflow uses the repository's install, test, and build scripts on a declared Node version. That contrast gives us something useful to test instead of a rule to memorize.

CI should reveal environmental assumptions, not introduce a second private build procedure. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

The happy path can hide a bad design: you can hide critical logic in long inline shell blocks that no one runs locally and still get one successful demo. Push past the demo. put reusable build logic in versioned project scripts and let YAML orchestrate those commands, then test the condition most likely to prove the choice wrong.

The module is moving toward one outcome: Create one workflow whose trigger, runner, permissions, commands, and failure behavior are obvious. Save the before-and-after evidence now; it will make the final project review much more honest.

Start from a clean local checkout and reproduce every workflow command in order. Save the evidence, then explain which requirement would force you to choose a different design.

Read contexts and expressions

Use event, repository, job, matrix, runner, needs, vars, and secrets data without confusing evaluation boundaries.

A deployment condition reads a trusted ref from context and passes a bounded value through env to the command. This is a small part of a GitHub repository whose workflow tests, builds, packages, and deploys a small site through a protected environment, but the boundary it creates affects everything that follows.

Expressions are evaluated by Actions, while shell variables and quoting are evaluated later by the chosen shell. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

Watch for one shortcut in particular: trying to splice untrusted event text directly into a shell script. It makes later failures harder to locate. Instead, distinguish expression and shell evaluation, pass data through environment, and quote it in the receiving shell. Prefer an implementation you can explain from the outside before optimizing it.

Keep the wider goal in view: Create one workflow whose trigger, runner, permissions, commands, and failure behavior are obvious. A short observation with concrete evidence is more useful than a confident sentence with no reproduction path.

Use a pull request title containing shell characters and prove the workflow treats it only as data. Hand the result to someone else and see whether the behavior is clear without an oral explanation.

Check your understanding: workflow foundations

Check events, workflows, jobs, steps, runners, contexts, expressions, and local reproducibility.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Test, build, and share data

Design job graphs, matrices, caches, artifacts, and failure evidence.

Design the job graph

Split work by independent outcome and connect jobs with explicit needs only where ordering matters.

Before choosing a tool or writing more code, make the requirement concrete. Lint and unit tests run together; the package job waits for both; deployment waits for the package and environment approval.

Jobs can run in parallel unless dependencies say otherwise, so the graph should reflect real data and release gates. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

A common failure is to put every command in one job or serialize unrelated work through unnecessary needs. The happy path may still work, which makes the mistake easy to miss. draw required outputs and gates, parallelize independent checks, and keep each job's purpose visible. Keep the first version small enough that every important input and output remains visible.

This supports the module goal: Build a fast CI graph that preserves useful evidence without sharing mutable state accidentally. Capture the request, command, trace, screenshot, test result, or other evidence that proves the result.

Measure a serial workflow, then reshape it into a dependency graph and compare total time and diagnostic clarity. Change one condition on purpose, predict the result, and compare the prediction with what actually happened.

Use a purposeful matrix

Test supported runtime or operating-system combinations without multiplying low-value jobs.

Start from the behavior you can observe. The library tests supported Node versions while the site build runs once on the production version.

A matrix should represent a compatibility promise or meaningful variation, not every available runner image. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

It is tempting to create a huge matrix that repeats identical evidence and exhausts minutes. That trades a clear decision for an assumption you will eventually have to debug. The stronger move is to tie each axis to documented support, use include or exclude deliberately, and keep one authoritative build. Make the boundary explicit in code and in the project notes.

Tie the decision back to the larger job: Build a fast CI graph that preserves useful evidence without sharing mutable state accidentally. Record enough evidence that another developer can repeat the result without relying on your memory.

Write the support policy first, then derive the smallest matrix that could falsify it. Repeat it once with an invalid or hostile condition and write down the boundary the failure revealed.

Separate caches from artifacts

Use dependency caches to save time and workflow artifacts to preserve outputs or evidence between jobs.

The package manager cache speeds installation, while the built site and test report are uploaded as artifacts. That contrast gives us something useful to test instead of a rule to memorize.

Caches are optimization keyed for reuse; artifacts are named outputs from one run with retention and download behavior. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

The happy path can hide a bad design: you can use a cache as the release artifact or assume a cache hit means dependencies are trustworthy and still get one successful demo. Push past the demo. key caches from the dependency lock, validate installs, and upload exact build outputs as immutable run artifacts, then test the condition most likely to prove the choice wrong.

The module is moving toward one outcome: Build a fast CI graph that preserves useful evidence without sharing mutable state accidentally. Save the before-and-after evidence now; it will make the final project review much more honest.

Restore a stale cache and prove the clean install corrects it; then download and inspect the build artifact. Save the evidence, then explain which requirement would force you to choose a different design.

Preserve failure evidence

Upload test reports, screenshots, logs, and summaries even when the main test step fails.

Browser-test screenshots and a machine-readable report upload under a bounded retention policy after failure. This is a small part of a GitHub repository whose workflow tests, builds, packages, and deploys a small site through a protected environment, but the boundary it creates affects everything that follows.

CI failures need enough evidence to diagnose without rerunning blindly or exposing sensitive data. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

Watch for one shortcut in particular: trying to dump every environment variable and raw request body into public logs. It makes later failures harder to locate. Instead, collect focused redacted evidence, use always-aware conditions carefully, and summarize the failing outcome. Prefer an implementation you can explain from the outside before optimizing it.

Keep the wider goal in view: Build a fast CI graph that preserves useful evidence without sharing mutable state accidentally. A short observation with concrete evidence is more useful than a confident sentence with no reproduction path.

Force a browser failure and diagnose it from the workflow page and artifacts without a local rerun. Hand the result to someone else and see whether the behavior is clear without an oral explanation.

Check your understanding: test, build, and share data

Check matrices, dependencies, caches, artifacts, outputs, service containers, and failure diagnostics.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Secure the pipeline

Minimize token permissions and protect secrets, forks, inputs, and action dependencies.

Minimize token permissions

Give GITHUBTOKEN no more repository access than each workflow or job requires.

Before choosing a tool or writing more code, make the requirement concrete. Pull request tests receive read-only contents access, while one release job receives the specific write permission it needs.

Workflow credentials are capabilities; broad write permission turns an injection or compromised action into a larger incident. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

A common failure is to accept broad default permissions because the token is temporary. The happy path may still work, which makes the mistake easy to miss. declare restrictive top-level permissions and grant narrow job-specific writes only at the trusted boundary. Keep the first version small enough that every important input and output remains visible.

This supports the module goal: Reduce workflow authority and prevent untrusted repository input from becoming privileged code execution. Capture the request, command, trace, screenshot, test result, or other evidence that proves the result.

Set permissions to empty or read-only, observe failures, and add back only the documented capability. Change one condition on purpose, predict the result, and compare the prediction with what actually happened.

Handle untrusted input and forks

Treat branch names, issue text, pull request metadata, changed code, and artifacts as attacker-controlled.

Start from the behavior you can observe. Fork code is tested without production secrets, and a trusted follow-up workflow consumes only verified outputs.

Repository events can combine untrusted content with secrets or write tokens, especially around fork workflows and privileged event types. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

It is tempting to check out and execute fork code in a context carrying repository secrets. That trades a clear decision for an assumption you will eventually have to debug. The stronger move is to separate untrusted execution from privileged work and validate every artifact or value crossing that boundary. Make the boundary explicit in code and in the project notes.

Tie the decision back to the larger job: Reduce workflow authority and prevent untrusted repository input from becoming privileged code execution. Record enough evidence that another developer can repeat the result without relying on your memory.

Threat-model a malicious pull request that changes tests, package scripts, artifact contents, and its title. Repeat it once with an invalid or hostile condition and write down the boundary the failure

revealed.

Protect secrets and cloud access

Scope secrets to environments and prefer short-lived federated credentials over long-lived cloud keys where supported.

Production deployment obtains a short-lived cloud credential through OIDC only after the protected environment is approved. That contrast gives us something useful to test instead of a rule to memorize.

Secret masking is not a permission boundary, and a secret exposed to a process can often be exfiltrated by that process. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

The happy path can hide a bad design: you can store one permanent administrator key as a repository secret and expose it to every job and still get one successful demo. Push past the demo. use environment scoping, least privilege, short-lived identity, rotation, and no secret access for untrusted code, then test the condition most likely to prove the choice wrong.

The module is moving toward one outcome: Reduce workflow authority and prevent untrusted repository input from becoming privileged code execution. Save the before-and-after evidence now; it will make the final project review much more honest.

Inventory every secret and replace one long-lived deployment credential with an identity-based flow or a narrower alternative. Save the evidence, then explain which requirement would force you to choose a different design.

Pin and review actions

Treat third-party actions as dependencies that execute with the job's workspace, network, and credentials.

Critical external actions are pinned to reviewed immutable commits and updated through a controlled dependency review. This is a small part of a GitHub repository whose workflow tests, builds, packages, and deploys a small site through a protected environment, but the boundary it creates affects everything that follows.

An action can run arbitrary code inside the job, so mutable tags and unknown maintainers create supply-chain risk. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

Watch for one shortcut in particular: trying to reference a convenient moving tag and grant the job broad permissions. It makes later failures harder to locate. Instead, minimize external actions, review source and ownership, pin immutable revisions, and update with evidence. Prefer an implementation you can explain from the outside before optimizing it.

Keep the wider goal in view: Reduce workflow authority and prevent untrusted repository input from becoming privileged code execution. A short observation with concrete evidence is more useful than a confident sentence with no reproduction path.

List every action in the workflow, its pin, maintainer, permissions, inputs, and update process. Hand the result to someone else and see whether the behavior is clear without an oral explanation.

Check your understanding: secure the pipeline

Check GITHUBTOKEN permissions, secrets, untrusted input, action pinning, forks, OIDC, and environment protection.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Reuse, release, and deploy

Reuse stable policy, control concurrency, promote artifacts, and protect environments.

Choose the right reuse boundary

Use project scripts, composite actions, or reusable workflows according to what must be shared.

Before choosing a tool or writing more code, make the requirement concrete. Several repositories call one deployment workflow with explicit inputs and inherited secrets only where required.

Reuse has layers: scripts share commands, composite actions share steps, and reusable workflows share jobs and policy. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

A common failure is to abstract one-off YAML immediately or copy security policy across repositories. The happy path may still work, which makes the mistake easy to miss. extract only stable repeated behavior and define inputs, secrets, permissions, versions, and ownership at the boundary. Keep the first version small enough that every important input and output remains visible.

This supports the module goal: Turn a verified build artifact into one controlled deployment with environment history and rollback. Capture the request, command, trace, screenshot, test result, or other evidence that proves the result.

Compare two workflows and choose the smallest reuse mechanism that removes meaningful duplication. Change one condition on purpose, predict the result, and compare the prediction with what actually happened.

Control concurrency

Cancel stale preview work and serialize production deployments without discarding the current safe release.

Start from the behavior you can observe. Preview runs cancel by branch, while production deploys share an environment concurrency group and do not cancel an active release blindly.

Multiple runs can race to deploy older code after newer code or consume the same mutable environment simultaneously. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

It is tempting to let every push race independently against the same production target. That trades a clear decision for an assumption you will eventually have to debug. The stronger move is to define concurrency from the resource being protected and choose cancellation behavior from rollback risk. Make the boundary explicit in code and in the project notes.

Tie the decision back to the larger job: Turn a verified build artifact into one controlled deployment

with environment history and rollback. Record enough evidence that another developer can repeat the result without relying on your memory.

Trigger three rapid commits and observe which test, preview, and production runs continue. Repeat it once with an invalid or hostile condition and write down the boundary the failure revealed.

Build a release from one artifact

Promote the exact tested artifact instead of rebuilding independently in the deployment job.

The package job uploads the static site with provenance data; staging and production download that same artifact. That contrast gives us something useful to test instead of a rule to memorize.

Build-once promotion preserves the connection between tests, review, and the bytes that reach an environment. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

The happy path can hide a bad design: you can run a fresh dependency install and build after approval, creating untested output and still get one successful demo. Push past the demo. produce one immutable artifact, record its source and digest, and promote it through environments, then test the condition most likely to prove the choice wrong.

The module is moving toward one outcome: Turn a verified build artifact into one controlled deployment with environment history and rollback. Save the before-and-after evidence now; it will make the final project review much more honest.

Compare artifact hashes in build, staging, production, and rollback records. Save the evidence, then explain which requirement would force you to choose a different design.

Use protected environments and rollback

Attach approvals, scoped secrets, deployment history, verification, and a tested rollback to production.

Production requires review, exposes only production credentials, records the deployment URL, runs a smoke test, and retains the previous artifact. This is a small part of a GitHub repository whose workflow tests, builds, packages, and deploys a small site through a protected environment, but the boundary it creates affects everything that follows.

An environment is both a target and a policy boundary around who and what may deploy. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

Watch for one shortcut in particular: trying to treat a green deploy command as proof that users received a healthy release. It makes later failures harder to locate. Instead, gate the environment, deploy, verify from outside, record the release, and roll back automatically or manually from known artifacts. Prefer an implementation you can explain from the outside before optimizing it.

Keep the wider goal in view: Turn a verified build artifact into one controlled deployment with environment history and rollback. A short observation with concrete evidence is more useful than a confident sentence with no reproduction path.

Deploy a deliberately broken health response to staging and rehearse the production rollback without rebuilding. Hand the result to someone else and see whether the behavior is clear without an oral explanation.

Check your understanding: reuse, release, and deploy

Check reusable workflows, composite actions, concurrency, releases, environments, approvals, deployments, and rollback.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Operate and improve CI/CD

Make failures actionable, control flakiness and cost, and rehearse recovery.

Make failures actionable

Use summaries, annotations, logs, artifacts, and ownership so a red run explains the next step.

Before choosing a tool or writing more code, make the requirement concrete. The summary names the failed gate, links evidence, states whether deployment occurred, and points to the owning team or runbook.

A CI signal loses value when failures are noisy, unactionable, or routinely ignored. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

A common failure is to print thousands of undifferentiated log lines and rely on someone rerunning the job. The happy path may still work, which makes the mistake easy to miss. surface the first useful cause, preserve focused evidence, and provide an owner and recovery instruction. Keep the first version small enough that every important input and output remains visible.

This supports the module goal: Operate the pipeline as a production system with useful signals, bounded cost, maintenance ownership, and recovery. Capture the request, command, trace, screenshot, test result, or other evidence that proves the result.

Give another developer only the workflow page and see whether they can diagnose three induced failures. Change one condition on purpose, predict the result, and compare the prediction with what actually happened.

Treat flaky tests as defects

Measure, quarantine narrowly, assign ownership, and fix nondeterminism instead of normalizing reruns.

Start from the behavior you can observe. The pipeline records retries for diagnosis but does not convert an unstable failure into silent success.

A flaky required check teaches people that red can mean nothing, weakening the entire release gate. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

It is tempting to add unlimited automatic retries and call the workflow reliable. That trades a clear decision for an assumption you will eventually have to debug. The stronger move is to track flake rate, preserve first-failure evidence, isolate the smallest unstable test, and set a repair deadline. Make the boundary explicit in code and in the project notes.

Tie the decision back to the larger job: Operate the pipeline as a production system with useful signals, bounded cost, maintenance ownership, and recovery. Record enough evidence that another developer can repeat the result without relying on your memory.

Introduce one timing-dependent test, measure its behavior, and replace time guessing with a deterministic condition. Repeat it once with an invalid or hostile condition and write down the boundary the failure revealed.

Control minutes, storage, and runners

Measure job duration, matrix value, cache hit rate, artifact retention, and runner exposure.

The project removes a redundant matrix axis, shortens artifact retention, and keeps untrusted fork code off persistent privileged runners. That contrast gives us something useful to test instead of a rule to memorize.

CI/CD consumes compute and stores data; self-hosted runners also create a persistent trust boundary. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

The happy path can hide a bad design: you can optimize only by adding caches or run all work on one powerful self-hosted machine and still get one successful demo. Push past the demo. profile the workflow, remove low-value work, bound stored data, and isolate runner trust according to the event, then test the condition most likely to prove the choice wrong.

The module is moving toward one outcome: Operate the pipeline as a production system with useful signals, bounded cost, maintenance ownership, and recovery. Save the before-and-after evidence now; it will make the final project review much more honest.

Produce a cost and risk report for each job, cache, artifact, and runner type. Save the evidence, then explain which requirement would force you to choose a different design.

Finish the pipeline review

Run the complete repository path from pull request through deployment, failure, cancellation, rollback, and credential revocation.

The final evidence maps triggers, permissions, artifacts, environments, deployments, logs, limits, owners, and runbooks. This is a small part of a GitHub repository whose workflow tests, builds, packages, and deploys a small site through a protected environment, but the boundary it creates affects everything that follows.

A pipeline is complete when its trust boundaries and recovery are tested, not when the happy path shows green. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

Watch for one shortcut in particular: trying to add more automation while nobody can explain token authority or restore the previous release. It makes later failures harder to locate. Instead, review the pipeline as code and infrastructure, exercise every critical failure path, and record remaining risks honestly. Prefer an implementation you can explain from the outside before optimizing it.

Keep the wider goal in view: Operate the pipeline as a production system with useful signals, bounded cost, maintenance ownership, and recovery. A short observation with concrete evidence is more useful than a confident sentence with no reproduction path.

Use one final scenario that includes a fork PR, failed test, rapid pushes, approval, bad staging result, fixed release, and rollback. Hand the result to someone else and see whether the behavior is clear without an oral explanation.

Check your understanding: operate and improve ci/cd

Check observability, flaky tests, costs, retention, runner trust, incident response, and the final pipeline review.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/github-actions-cicd/.