

THE ENGINEERING SERIES / OPEN EDITION 0.1

MODERN CI

Building a Reliable CI/CD Platform

Go · Angular · PostgreSQL
Independent Git repositories

Aleksandar Popovic

24 chapters · Practical laboratory · Operational field guide

October 2026 · Text CC BY 4.0 · Code MIT

Contents

Preface	3
1. Think in runs, not shell scripts	6
2. Draw control and trust boundaries	9
3. Make every application an independent repository.....	12
4. Keep contracts and dependency resolution honest.....	15
5. Run the laboratory before designing the fleet	18
6. Model identity, ownership, and permission together	21
7. Compile a pipeline into an immutable graph	24
8. Commit requests before acknowledging events.....	27
9. Claim work with short PostgreSQL transactions.....	30
10. Fence attempts and preserve uncertainty.....	33
11. Design an outbound runner protocol	36
12. Execute containers within explicit limits.....	39
13. Move live logs without hiding gaps	42
14. Preserve artifacts and keep cache trust separate	45
15. Build once and preserve the image digest.....	48
16. Integrate source events, schedules, and matrices.....	51
17. Give credentials to an authorized attempt, not a script name	54
18. Deploy an approved immutable release.....	57
19. Recover from cancellation, uncertainty, and rollback.....	60
20. Build an operator interface that explains state	63
21. Keep the CLI precise and extensions constrained	66
22. Observe reliability, capacity, and recovery	69
23. Test failures and upgrade through compatible states	72
24. Publish a reproducible release and maintain the project.....	75
Appendix A. Laboratory runbook.....	78
Appendix B. From chapters to eighty implementation work items	81
Appendix C. Acceptance gates and failure experiments.....	88
Appendix D. Glossary	91
Appendix E. Primary sources	95

Preface

A CI/CD system is easy to demonstrate and difficult to trust. Starting a command is a small part of the job. The platform must know which source revision it is running, who authorized the work, where credentials may be used, whether a disconnected worker is still changing a target, and which result belongs to the current attempt. A useful interface makes these distinctions visible instead of compressing them into a green or red badge.

This book develops an original, self-hosted-first platform named **Modern CI**. The target is not a cosmetic wrapper around Jenkins. It has its own control plane, scheduler, runner protocol, pipeline compiler, web application, and command-line client. Nor does it promise compatibility with every historical plugin or dynamic pipeline program. The intended first release serves teams with trusted repositories and dedicated Linux runners. Its scope is to build, test, publish artifacts, and coordinate controlled deployments without requiring Kubernetes.

The architecture uses Go for the server, scheduler, runner, and CLI; Angular with Tailwind for the browser; PostgreSQL for durable state; Docker and BuildKit for execution and image construction; and local or S3-compatible storage for binary objects. Each application has its own repository. Shared domain libraries and wire contracts have two additional repositories. A superproject pins the combination that has been tested together.

A book, a laboratory, and a product design

These are three different deliverables. The manuscript explains the engineering decisions. The companion laboratory contains small, runnable Go examples for graph compilation, attempt fencing, idempotent results, webhook signatures, bounded log segments, and approval identity. The production design describes the additional persistence, identity, infrastructure, and operational work required to turn those ideas into a platform.

The laboratory is deliberately in memory. It does not start containers, provide a production API, connect to PostgreSQL, or implement an Angular application. SQL and pipeline files are separate teaching examples. Their presence is not evidence of a deployed product. The verification directory records which checks were actually executed for this edition. Production claims require the acceptance gates described later in the book.

Intended reader

You should be comfortable reading Go, SQL, HTTP requests, and shell commands. Experience maintaining build servers will make the failure scenarios familiar. The text explains the domain rather than teaching Go or Angular from the beginning. Chapters about privileges assume that you control the systems being configured. Use disposable infrastructure for experiments and do not give the laboratory production credentials.

Read Chapters 1–4 to understand the boundaries, then run Chapter 5. Chapters 6–10 establish identity, compilation, transactions, and ownership. Chapters 11–17 turn those records into controlled execution. Chapters 18–21 cover deployment and operator interfaces. Chapters 22–24 explain operations, upgrades, and release management. The appendices provide a laboratory runbook, an eighty-work-item implementation map, acceptance gates, a glossary, and primary references.

How examples are labeled

A **laboratory command** runs the included Go module. A **design example** explains an interface or schema that a complete implementation must build. An **operator procedure** describes actions to adapt and review on a deployment you own. Example repository names and resource identifiers are not evidence that those resources already exist.

The source files under `manuscript/` are the canonical edition. The website, PDF, EPUB, and reading sample are built from those files. Corrections should be made there, accompanied by a test or a reproducible explanation when behavior changes. The book repository is independent of the eight application/workspace repositories described in Chapter 3. Reading the book does not require cloning them.

Open licenses and attribution

Copyright (c) 2026 Aleksandar Popovic. Original prose, exercises, answers, and original publication styling are offered under Creative Commons Attribution 4.0 International. Original code, snippets, scripts, and configuration examples are offered under the MIT License. The repository contains the licensing map and MIT notice. Third-party dependencies keep their own licenses. The Creative Commons license permits sharing and adaptation with its attribution conditions; the MIT notice governs reuse of the code. [S23](#) [S24](#)

Modern CI is a working project title, not a trademark-clearance claim. References to Jenkins, Go, Angular, PostgreSQL, Docker, GitHub, and other projects identify the technologies being discussed; they do not imply affiliation or endorsement. There are no measured product benchmarks or customer claims in this edition.

Evidence before confidence

A unit test of a lease model does not prove that two database transactions coordinate correctly. A valid configuration file does not prove that a container runtime enforces it. A local book build does not prove that a hosted publication has been deployed. Throughout the book, the question is not merely “does this look plausible?” but “which observation would make this claim defensible?”

That habit is the real foundation of a reliable CI/CD platform.

1. Think in runs, not shell scripts

The misleading first demonstration

Imagine a web page with a repository selector and a Run button. The server clones a project, invokes a shell, and streams its output to the browser. A successful test suite produces a green badge. This is a useful demonstration, but it hides the hardest questions. What happens when the browser disconnects? Does restarting the server kill the command? Can a second click deploy twice? Which commit was cloned when the branch moved between the click and the checkout?

Start the design with records that survive these questions. A **definition** describes the work. A **run** is a particular request to execute an immutable definition against specific inputs. A **job** is a schedulable node in the graph. A **step** is an operation within that job. An **attempt** is one concrete execution of a job. A **deployment** is a separately tracked change to an environment. These terms let an operator distinguish a user's intention from a worker's observations.

Jenkins already has pipeline-as-code, durable pipeline execution, parallel work, and approval steps. Reimplementing those names is not a product advantage by itself. Its documentation is a useful reference for the existing concepts. Modern CI's design goal is a coherent contract and an understandable operational experience, not a claim to have invented pipelines. [S01](#)

Follow one request through the system

A developer requests a run for source commit `c7`. The server authorizes the request and freezes its inputs. The compiler creates test, package, and deploy jobs with explicit dependencies. The scheduler makes test eligible. A runner receives attempt `a1`, reports its start, uploads bounded log segments, and submits a result. Only a committed successful result makes package eligible.

If `a1` fails because the execution host disappears, a retry does not erase it. The platform creates `a2`, preserving the first attempt and its evidence. A retry policy decides whether the failure is eligible. The job is the logical unit whose eventual outcome can satisfy a dependency; the attempt is the physical work whose history must remain inspectable.

Use identifiers rather than human names in internal relationships. Names are useful labels but can change. A deployment called `production` must still point to a specific environment record and artifact digest, even if someone renames a project halfway through the run.

Desired, observed, and committed state

An operator can request cancellation without a worker having stopped. A worker can exit before its final result is committed. A target can receive a deployment before the network reply reaches the runner. Avoid turning these into one ambiguous status field.

A useful run detail page might show:

```
Requested action: cancel
Attempt: a2
Last observed execution state: running
Control state: cancel requested
Last heartbeat: recorded server timestamp
Target side effects: not yet reconciled
```

This is less visually tidy than immediately displaying “cancelled,” but it is more truthful. An uncertain state is a supported outcome, not an embarrassing exception to hide.

Terminal results should be immutable within an attempt. Duplicate identical reports can be acknowledged. A conflicting later report should produce a conflict record, not overwrite the first committed outcome. Operator correction can create a separately authorized reconciliation event while preserving what the original worker said.

Define success before breadth

The first useful milestone is one repository, one dedicated runner, an immutable checkout, a bounded command, durable result history, and a readable log. Add an artifact next. Add an approval-controlled deployment only after execution uncertainty has an explicit model. This path puts correctness before a plugin marketplace or a sophisticated visual editor.

Write success criteria as observable behavior. “Supports retries” is weak. “Creates a new attempt, preserves the earlier log, and rejects the earlier attempt's late result” is a testable requirement. “Secure secrets” is weak. “A project reader cannot retrieve plaintext, and a revoked attempt cannot redeem a credential” is a stronger starting point.

Exercise

A user presses Retry while the previous attempt has lost contact with the server. A developer proposes replacing the old attempt row with a fresh running state. Identify three pieces of evidence that this destroys and the safer behavior for a deployment job.

Worked answer

The replacement loses the old attempt identity, its execution timeline, and the association between its output and a particular claim. It also makes a late result difficult to distinguish from the new execution. Keep the attempt and mark its execution status uncertain. For a deployment, retain the target lock and reconcile the external state before authorizing another attempt. A new attempt is appropriate only after the retry policy and target-safety checks permit it. The old attempt remains part of the record.

Completion evidence

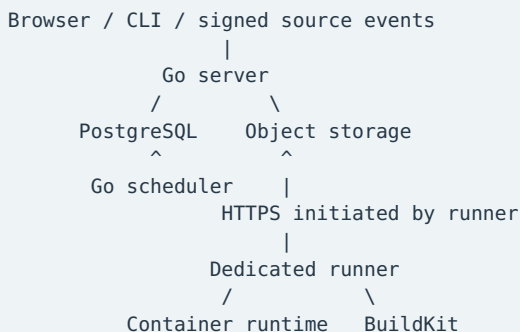
A domain review should be able to explain every identifier and every transition in the request-to-result path. No API field should equate a cancellation request with confirmed termination.

2. Draw control and trust boundaries

Separate authority from execution

Modern CI has five applications: server, scheduler, runner, web, and CLI. The server handles user-facing APIs, identity, and integrations. The scheduler evaluates durable work. The runner performs authorized execution on a separate host. The web and CLI present the same control API through different interfaces. Shared code is not a sixth network service.

The server and scheduler must not execute repository commands. They do not need a build-host Docker socket. This follows a boundary already emphasized in Jenkins controller-isolation guidance: code that participates in a build must not inherit the controller's filesystem and administrative privileges. [S02](#)



This drawing describes authority, not a requirement for one machine per box. The server and scheduler can initially run as different processes on the same control host. PostgreSQL can be managed or self-hosted. What matters is that build execution is on a deliberately separate trust surface with controlled access back to the API.

One database, different responsibilities

For the first release, PostgreSQL is authoritative for runs, jobs, attempts, permissions, leases, schedules, and audit events. Object storage holds bytes: logs, reports, and artifacts. The scheduler must not infer truth from a browser connection or from an in-memory event queue. Notifications merely wake it to reread committed state.

The server exposes an explicit migration command. A single migration tree lives in the shared core repository. Scheduler startup checks schema compatibility; it does not race another application's migration runner. This avoids independent repositories becoming independent and conflicting owners of the same database.

Avoid a generic “execute command” endpoint on the control plane. The runner receives a constrained execution description after authorization. The runner API accepts only messages tied to its current identity and authorized attempt. A compromised runner should not be able to turn the server into a filesystem browser or cloud administrator.

Classify code by trust

A signed webhook proves something about a delivery, not about whether the repository content is safe to execute. A developer with permission to change a test can cause arbitrary actions inside the test process. A private repository is not automatically trustworthy if contributors have different privileges. GitHub's secure-use guidance specifically describes risks around self-hosted runners and untrusted pull-request content. [S17](#)

The initial supported model is a team operating dedicated runners for trusted projects. Production deployment uses another pool, with a narrower target set and credentials unavailable to ordinary build jobs. Untrusted public contributions require a separately reviewed isolation mode. Do not quietly enable that mode because the same YAML syntax happens to work.

An outbound runner connection simplifies network exposure, but it does not make every server instruction safe. Authenticate both ends, validate the protocol, constrain the received fields, and reject unknown privileged options. Network direction and authorization are different controls.

Resource boundaries are security boundaries

A log upload, compressed artifact, report file, and matrix expansion can all consume unbounded resources unless the protocol sets limits. Record maximum request sizes, maximum uncompressed archive size, file count, graph size, and execution duration. A limit without a visible error becomes a debugging problem; a limit without server enforcement becomes a suggestion.

Separate the control plane's outbound destinations as well. A notification URL or repository URL must not become an unrestricted request proxy into metadata services or private administrative endpoints. Destination policy, redirects, DNS resolution, and credential forwarding need a design rather than one regular expression.

Exercise

A build pool and a production pool use the same machine but different labels. Build containers can reach the host Docker socket. Does this establish the boundary required by this chapter?

Worked answer

No. Labels describe eligibility, not isolation. A process controlling the host daemon may affect other containers and host resources. Docker documents the daemon's authority and the risks of exposing it. Separate deployment authority from build execution, remove the socket from job containers, and select an isolation design appropriate to the accepted code. Different labels on one uncontrolled host do not create separate trust zones. [S08](#)

Completion evidence

Produce a threat model that names who controls repository content, who owns runners, which credentials cross each boundary, and what happens if one runner is compromised. Every arrow in the architecture needs authentication, authorization, and a bounded payload contract.

3. Make every application an independent repository

Eight repositories compose one system

The application's source topology is intentionally explicit:

Repository	Workspace path	Responsibility
modern-ci-workspace	root	Integration, pins, operational composition
modern-ci-server	apps/server	HTTP API and identity
modern-ci-scheduler	apps/scheduler	Scheduling and reconciliation
modern-ci-runner	apps/runner	Execution and runner protocol
modern-ci-web	apps/web	Angular operator interface
modern-ci-cli	apps/cli	Command-line client
modern-ci-core	packages/core	Public domain/store/compiler libraries
modern-ci-contracts	packages/contracts	Wire schemas and client generation

The book you are reading belongs in a separate `modern-ci-book` repository. It does not pretend to contain these application histories. The companion module is a teaching fixture, not an eighth application hidden inside the book.

A Git submodule is a separate repository referenced by a gitlink in a superproject. The gitlink selects a commit; `.gitmodules` records configuration such as its path and URL. Copying a folder into a ZIP does not create that relationship. [S03](#)

Bootstrap in the right order

Create an initial commit in each child repository and make it reachable in a repository you control. Then register the children from the workspace. The following is an operator example after the child repository exists; replace the owner deliberately:

```
OWNER=your-github-owner
git submodule add \
  "https://github.com/$OWNER/modern-ci-server.git" \
  apps/server
git add .gitmodules apps/server
git commit -m "Register the server repository"
```

Repeat the pattern for the other components. Do not insert invented commit hashes into a lock file. Before publishing the parent, ensure readers have access to the child commits it references. A publicly readable workspace pointing to inaccessible private children is not a reproducible public release.

Cloning an existing workspace normally requires recursive initialization:

```
git clone --recurse-submodules REPOSITORY_URL
git submodule status --recursive
```

The checkout selected by the parent may be detached. Before editing inside a child, create or switch to an appropriate branch. Commit inside the child, then stage the updated gitlink in the parent. Git's submodule documentation explains the separate histories and update behavior. [S04](#)

Source separation is not build independence

A runner repository that imports a sibling directory through a relative replacement is not independently buildable. A Dockerfile in `apps/server` cannot rely on undeclared files above its build context. A web repository that needs a local symlink to another repository is not a complete source checkout.

Make each child own its dependency declarations, tests, Dockerfile where applicable, and release procedure. The root owns the tested combination, not a hidden source overlay. Root integration tests are necessary, but they cannot replace a standalone build of every application.

The shared core package must expose intentional public APIs. An internal package in one Go repository is not a convenient cross-repository import surface. Keep low-level internals private and publish narrowly named packages for the compiler, queue transactions, domain types, and store. The contracts repository should remain independent of persistence implementation.

Coordinate a change without losing history

Suppose a new job field requires schema and UI changes. Start with the contracts and compatibility fixtures. Update the core validation. Update server and scheduler consumers. Regenerate the web client with a receipt naming the schema digest and generator. Test each changed repository independently, then test the combined workspace.

Commit the children first. Update the workspace gitlinks second. Record a source lock from those exact commits. A release manifest later adds built image digests and schema/protocol compatibility. The source lock and release manifest are different records: one identifies code, the other identifies deployable outputs.

Do not force-rewrite commits already used by a release or evidence record. Use forward changes and a new workspace snapshot. Rewriting a child history can invalidate references preserved in several other repositories.

Exercise

A developer changes the runner, commits only the workspace, and sends the workspace commit for review. The reviewer gets the old runner. Why, and how should the change be repaired without inventing a new history?

Worked answer

The parent cannot store uncommitted source inside a submodule as an ordinary directory patch. Commit the runner changes inside its repository, publish that child commit through the normal review path, then commit the new gitlink in the workspace. Verify the resulting workspace from a fresh recursive clone. The important evidence is not that the author's working tree looked correct, but that another checkout can retrieve the same child commit.

Completion evidence

A clean consumer environment can clone the workspace recursively, identify all seven child commits, and build each application without relying on untracked sibling files.

4. Keep contracts and dependency resolution honest

Three kinds of compatibility

A source commit, a library version, and a network protocol version answer different questions. The workspace pins source. A Go module declaration identifies an import dependency. A runner handshake declares which messages an installed binary understands. Pinning the first does not automatically establish the other two.

The contracts repository owns the HTTP schema, runner wire format, pipeline grammar, and generated client descriptions. The core repository owns validation semantics and the single migration tree. The server and scheduler share the core through versioned dependencies. Runner and CLI depend on wire contracts without importing the database implementation.

Model compatibility as an explicit table: server version, protocol version range, schema range, generated client revision, and tested runner versions. Reject unsupported combinations early. A helpful failure says which range is required; it does not let a worker begin a job and discover an incompatible field halfway through execution.

Use a local workspace without hiding release defects

A Go `go.work` file can combine multiple modules during development. It is an overlay for collaborating on sibling modules, not a promise that an isolated consumer can resolve those dependencies. The Go workspace tutorial and module reference describe that distinction and module version resolution. [S05](#) [S06](#)

For this project, generate an ignored root `go.work` after the child modules exist. Check the integrated workspace, then check applications with workspace resolution disabled:

```
cd apps/server
GOWORK=off go test ./...
GOWORK=off go build ./cmd/server
```

These are target-application procedures, not commands provided by the book's laboratory. They should run in a fresh checkout with only the declared dependencies. Do not ship replace directives pointing to `../../packages/core` and call the result independent.

Before dependency repositories are published, choose a clearly labeled local verification path. One option is a verified vendor snapshot prepared from pinned source, with a receipt describing its producer commits and checksums. Another is a temporary local module proxy with valid module versions. Neither path proves that public module resolution works. The release gate must test the actual published dependency path separately.

Generate clients with receipts

The browser can commit a generated TypeScript client into its own source tree for the first release. This avoids forcing a registry publication for each local schema iteration while keeping the web checkout self-contained. Record the full contracts commit, schema content hash, generator identity, generator version, and relevant flags.

A generated file with no provenance soon becomes hand-maintained by accident. Regenerate in integration CI and fail on drift. Do not patch generated DTOs to make a UI build pass while the server still advertises a different shape. Place presentation types outside the generated directory.

For wire evolution, prefer additive optional fields with defined defaults. An unknown enum value may need an “unsupported” presentation instead of a crash. Removing a required field or changing its meaning needs a versioned migration path. A protocol can be syntactically parseable and still semantically incompatible.

One owner for schema change

Let the server's explicit administration command apply migrations from the core package under an exclusive migration lock. Normal API and scheduler startup check a supported range. They should not independently modify schema while accepting work.

An expansion migration can add a nullable column before code begins using it. Backfill in bounded batches. Only after the new path is proven should a later release remove the old representation. Keep both protocol and database changes reversible within a documented window, and state where downgrade ceases to be safe.

Avoid an all-purpose compatibility claim. Independently released repositories do not imply that arbitrary versions can be combined. The compatibility matrix is evidence for particular combinations, not a universal property of the architecture.

Exercise

The web builds successfully using a sibling SDK symlink, and the server builds successfully using `go.work`. Can the workspace be released? State the missing checks.

Worked answer

Not on those observations alone. Build the web from its own checkout with its committed lockfile and generated client receipt. Build Go consumers with `GOWORK=off` from resolvable module versions or a documented, verified vendor snapshot. Then validate the actual release resolution path and run protocol/schema compatibility tests against the built binaries. Integrated source convenience is useful, but it must not substitute for independent dependency resolution.

Completion evidence

Every published component includes a dependency record and a tested compatibility range. A build must reveal its source and dependency inputs without requiring the author's working directory.

5. Run the laboratory before designing the fleet

A small executable model

The companion code under `examples/lab` is a standard-library-only Go module. It provides deterministic graph validation, an in-memory attempt ledger with fencing, webhook HMAC validation, a bounded log-segment store, and exact approval-subject matching. It does not execute arbitrary commands. That limitation lets us explore ownership and replay without accidentally building an unsafe public executor.

Run these laboratory commands from the book repository:

```
cd examples/lab
go test ./...
go test -race ./...
go run ./cmd/demo
```

The demonstration creates a graph, starts an attempt, makes it uncertain after a simulated lease expiry, authorizes a retry only after a stopped observation, and rejects a stale completion from the old attempt. It also shows a log-upload retry being recognized without duplicating bytes. The tests state the precise expected outcomes rather than relying on the prose output alone.

Use a supported Go release for ongoing work. The module's language baseline is not a security-support promise. The verification report records the compiler actually used for this edition. No third-party Go dependencies are downloaded by the examples.

Time is an input in the model

The ledger accepts an explicit timestamp. Tests can move from t_0 to $t_0 + 31$ seconds without waiting in real time. This makes expiry tests fast and deterministic. Production ownership, however, uses database-coordinated time and atomic updates. A unit test of supplied timestamps does not validate host-clock behavior or PostgreSQL transaction isolation.

The in-memory ledger uses a mutex to demonstrate serialized decisions within one process. That mutex is not a distributed lock. Restarting the process loses its state. Chapter 9 explains the database transaction that a durable version requires, and Chapter 23 lists the live concurrency tests that are still necessary.

Keep syntax and semantics separate

The DAG input to the laboratory is a list of job identifiers and dependency lists. It intentionally does not parse YAML. A complete pipeline frontend needs strict schema validation, duplicate-key handling, bounded input size, and carefully specified scalar behavior before it calls the graph compiler.

The demonstration teaches the graph invariant: a job can depend only on known, distinct jobs; self-dependencies and cycles are rejected; the compiled order is deterministic. It does not establish that an arbitrary YAML parser is safe for untrusted input.

Similarly, the signature helper works on exact byte slices and a signature string. It is not an HTTP server. A production endpoint must limit body size, enforce content type, persist the delivery receipt, and apply repository authorization. Signing tests are necessary but do not perform those other jobs.

Inspect the code instead of trusting the demo

Read `dag.go`, then the tests for missing dependencies and cycles. Read `ledger.go`, then the tests for expired leases, stale identities, cancellation, and duplicate results. Read `logstore.go`, where the next segment is accepted only if its sequence matches the expected value and its content fits the configured quota. Read `approval.go`, where matching a human-readable release label is deliberately insufficient.

Change one behavior and run the tests again. For example, remove the fence comparison from result acceptance. The stale-result test should fail. Put it back before continuing. A demonstration is educational only when it is sensitive to the defect it claims to prevent.

Expected boundaries

The laboratory has no persistent queue, cloud credential exchange, container sandbox, browser session, production secret vault, or SSH deployment. It uses synthetic identifiers and loopback-free pure functions. The separate SQL files are reviewable design examples, not an automatically provisioned database.

Keep the book's implementation map as a set of future work items. A passing laboratory suite completes the examples' checks, not all eighty product work items. This distinction prevents an attractive repository from overstating what can actually be operated.

Exercise

The race detector reports no issue for the in-memory ledger. A teammate concludes that running three schedulers against PostgreSQL is safe. What experiment should come next?

Worked answer

Run multiple actual database clients claiming the same eligible row under the proposed transaction. Assert that exactly one claim becomes current, other claimers select different rows or receive no work, and stale completions affect zero current rows. Repeat with transaction aborts, statement timeouts, connection loss after commit, and process termination. The race detector helps find memory races in the Go process; it does not exercise database locking or network uncertainty.

Completion evidence

Save the test command, compiler version, exit status, and output. Record the laboratory's scope explicitly. A failed prerequisite is a blocker to the corresponding experiment, not a reason to report it as passed.

6. Model identity, ownership, and permission together

Scope every resource

An organization contains projects; a project owns pipelines, runs, artifacts, and secret-use policies. Runner pools and environments have explicit access grants. A request is authorized against both the action and the object being accessed. Hiding a button in Angular is not an authorization rule.

Use immutable organization and project identifiers in ownership relationships. Where practical, include them in compound foreign keys or lookup predicates so an accidental unscoped query is difficult to write. A run lookup should not retrieve a globally matching identifier and only sometimes check its organization afterward. Tests must deliberately attempt cross-project reads, writes, downloads, and websocket or streaming subscriptions.

Roles are named bundles of capabilities, not special branches scattered through the code. A viewer can inspect permitted history. A developer can author and request builds. An operator can manage runner availability. A release approver can authorize a particular environment transition. An administrator manages membership. Some organizations may combine these roles, but the model should not require them to be identical.

Permission to write code is permission to influence execution

A person who changes build commands can cause actions using whatever the job receives. Treat secret use and privileged runner selection as separate permissions from pipeline editing. The server evaluates them after compiling the requested configuration and again before dispatch when revocations matter.

The YAML field `pool: production` is a request, not an entitlement. An authorized pool record must belong to or explicitly be shared with the project. Labels such as `linux`, `arm64`, and `gpu` further narrow eligible machines only after this authorization step. A label must never confer access to another tenant's host.

A shared runner pool needs rules for what happens when grants change. Already issued external credentials may remain valid until expiry. New claims and credential redemptions can be refused immediately. Make that distinction visible to administrators so “revoke” is not presented as retroactive erasure of all authority.

Separate credential classes

A browser session, CLI token, one-use enrollment token, and long-lived runner identity are different credential classes. Do not accept one where another is expected. Give them different audiences, scopes, expiry policies, and storage conventions.

For browser cookies, use secure transport, `HttpOnly`, an intentional same-site policy, and a CSRF defense for state-changing operations. Avoid logging cookie headers or returning session secrets to frontend diagnostics. For CLI authentication, store credentials in a restricted local file or platform credential store, and keep normal command output free of tokens. These are proposed application controls, to be verified in its HTTP and packaging tests.

Runner enrollment should be a one-time action authorized for a particular pool. Exchange the enrollment credential for a revocable identity, then remove the enrollment secret. A runner rotation procedure must handle an overlap window deliberately; otherwise a routine key change can look like an unexplained loss of all workers.

Audit decisions, not just mutations

An audit event should answer who requested the action, which resource was involved, which policy was used, and what decision was committed. Store actor and resource identifiers, request correlation, and a safe summary. Do not store plaintext secrets, full authorization headers, or uncontrolled payloads merely because they are useful during debugging.

Sensitive actions such as changing an environment policy, issuing a runner enrollment token, or reconciling an uncertain deployment deserve specific event types. A generic “settings changed” event leaves operators guessing during an incident.

Authorization tests should run at the service boundary as well as the HTTP boundary. A future background task or CLI adapter must not bypass checks simply because it calls the same underlying function without passing through a web middleware layer.

Exercise

A developer may edit a staging pipeline and a project contains a production credential. The interface never displays that credential's value. Is the credential protected if the developer can add it to the staging job's environment?

Worked answer

No. Code receiving the credential can use it or send it elsewhere, even if the interface is write-only. Enforce secret-use grants separately from secret visibility and pipeline editing. Restrict the production credential to an authorized environment and execution context, and recheck that policy at redemption. Log masking can reduce accidental disclosure but cannot safely grant a secret to malicious code.

Completion evidence

Maintain an authorization matrix and negative tests for every resource family, including streaming endpoints, signed downloads, credential redemption, and runner pool selection.

7. Compile a pipeline into an immutable graph

Treat configuration as a small language

A pipeline file is an input language, not a trusted object. Give it a version, a strict schema, and predictable error locations. Reject unknown keys where silently accepting a typo would change execution. Bound file size, nesting, parameter count, and graph expansion. Define the meaning of strings, numbers, and booleans rather than depending on incidental parser coercions.

The following design example illustrates the intended syntax. It is not a file consumed by the included laboratory and is not a Jenkins or GitHub Actions configuration:

```
apiVersion: ci/v1
name: library-check
jobs:
  test:
    pool: linux-build
    toolchain: go-supported
    steps:
      - run: go test ./...
  package:
    needs: [test]
    pool: linux-build
    toolchain: go-supported
    steps:
      - run: go build ./...
```

The toolchain alias is resolved to an allowed immutable image identity when the run is created. The frozen plan records that identity. Changing the alias later must not change an already compiled run.

Build a bounded DAG

The graph compiler first creates a node table. Each identifier must be unique. It then verifies that every dependency exists, is distinct within its list, and is not the node itself. Finally, it performs cycle detection and emits a deterministic topological order. Sorting equally eligible nodes makes tests and plan hashes stable; execution can still run independent nodes concurrently.

A graph with test -> package -> deploy means that successful dependencies enable a later job. It does not imply that files automatically appear on another worker. Declare artifact edges separately. An environment approval may hold a job after all computational dependencies have succeeded.

Conditional behavior needs a small, safe expression model. Do not evaluate arbitrary Go, JavaScript, shell, or template functions while compiling. Begin with explicit comparisons against typed parameters and recorded source metadata. Secrets must not participate in a plan hash through their plaintext values, nor appear in compilation diagnostics.

Freeze the decision inputs

A run snapshot should include the requested source commit, configuration content digest, compiler version, resolved toolchain identities, parameter values after validation, policy references, and the compiled graph. Store secret references rather than secret values. Record which policy decisions remain live and will be rechecked before execution.

This last distinction matters. The graph is immutable, but a revoked credential or disabled environment cannot be ignored merely because the earlier snapshot was valid. Freeze execution intent while allowing authority to become more restrictive. A policy change that expands authority should not silently add capabilities to a pending run.

A user-facing editor and a freestyle form should compile into the same intermediate representation. The form is a convenient authoring surface, not a second execution engine. Exporting and importing supported configurations should preserve semantics. Unsupported constructs should produce explicit errors instead of being dropped during a visual-editor save.

Plan before scheduling

Store a failed compilation as a clear validation outcome rather than creating a partially runnable graph. A run cannot begin execution until its plan is committed in full. Hash the normalized plan using a specified encoding; do not hash an unordered map serialization whose output may change between runs.

Matrix expansion belongs before execution too. Calculate the resulting node count before allocating the complete product. A three-dimensional matrix with twenty values per axis has eight thousand combinations. Apply limits before the expansion consumes resources, and require a deliberate policy for failed matrix members and aggregate status.

The laboratory tests the graph invariant and deterministic order. Schema parsing, conditional evaluation, matrix limits, and frozen-plan storage are product work beyond that helper.

Exercise

A job references needs: [tests], but the defined job is test. A proposed convenience feature treats missing dependencies as optional. What operational failure could this cause?

Worked answer

The package or deployment job could start without the intended tests, producing a success signal for work that never ran. Reject the missing dependency with a location-aware error and do not commit a runnable graph. Optional behavior should have explicit syntax and a documented aggregation rule, not arise from an unresolved name.

Completion evidence

Compiler tests cover duplicate identifiers, missing dependencies, cycles, self-dependencies, unknown fields, expansion limits, deterministic plans, and policy changes between compilation and dispatch.

8. Commit requests before acknowledging events

A receipt is part of the transaction

A source webhook may arrive twice, a manual request may be retried after a timeout, and a browser may repeat a submission. Design idempotency as durable data. An in-memory set of recently seen delivery identifiers disappears on restart and diverges across API instances.

For a manual run request, require a project-scoped idempotency key. Record a digest of the normalized request with the response identity. An identical retry returns the existing run. The same key with a different payload returns a conflict. Without the digest, a client can accidentally reuse a key and receive a response to a different request.

For a provider event, use a uniqueness boundary such as provider, installation, and delivery identifier. The event receipt and its durable processing instruction must be committed before acknowledging accepted work. Parsing and authorization failures can receive explicit responses without creating a run.

Verify exact bytes

GitHub webhook validation uses a secret-derived HMAC signature, supplied in X-Hub-Signature-256. Verification must operate on the raw request body and use a timing-safe comparison; decoding and reserializing JSON changes the signed bytes. The included helper demonstrates only this byte-level check. [S15](#)

Limit the request body before loading it into memory. Verify content type and expected event kind. Treat the installation and repository identities as untrusted fields until tied to the configured integration. Do not fetch a user-controlled URL from the payload simply because the signature is valid.

Signature validation is not deduplication. A correctly signed payload can be delivered again. GitHub recommends webhook secrets, HTTPS, prompt responses, and identifying deliveries; our design combines those delivery practices with a durable inbox. [S16](#)

Inbox, run, and outbox

One transaction can insert the receipt, create the frozen run, and append an outbox entry describing a notification or scheduling wakeup. If compilation is too expensive for the request path, first commit a durable inbox job, acknowledge it, and let a worker perform the rest. Do not acknowledge an event that exists only in a process-local queue.

An outbox dispatcher reads committed records and sends external notifications. It marks completion after a successful response according to the adapter's contract. A network failure after the remote service acted is ambiguous, so retry with a provider-supported idempotency key where available. Otherwise, tolerate a duplicate notification rather than falsely promising exactly-once delivery.

The scheduler always rereads database state. Losing a wakeup does not lose a run; it merely delays discovery until the next scan. Periodic recovery makes the event transport an optimization instead of a second source of truth.

Avoid duplicate intent and preserve distinct intent

Deduplication must not collapse two genuinely different source events. A push event and a manually requested rebuild for the same commit may be distinct, legitimate runs. Use event identity and request identity, not source commit alone, as the deduplication key.

Policy can still suppress redundant work intentionally, such as cancelling an older pending branch run when a newer commit arrives. That is a separate, audited scheduling decision. It must not erase the history of a run that has already started deploying.

Provider status updates should name the exact run and commit they represent. A delayed result for an older branch head must not replace the visible status of a newer run through an imprecise lookup such as “most recent project build.”

Exercise

An API instance inserts a run, sends a notification, and then commits the transaction. The transaction aborts. What do users see, and how should the flow change?

Worked answer

Users may see a notification referring to a run that never existed durably. The notification can also trigger dependent behavior outside the database. Insert the run and an outbox record within the same transaction, commit, and let a dispatcher send the notification afterward. The dispatcher needs its own idempotency and recovery policy because commit and external delivery cannot be made atomic merely by placing them near each other in code.

Completion evidence

Test duplicate deliveries across two API instances, conflicting idempotency keys, an abort before commit, a lost response after commit, and a dispatcher crash after sending but before recording completion.

9. Claim work with short PostgreSQL transactions

Claiming is a state transition

A durable queue is not just a table with `status = pending`. Claiming must atomically select eligible work, assign its current owner, create an attempt, and reserve the relevant capacity. If these are separate uncoordinated operations, two schedulers can start the same work or exceed a shared limit.

PostgreSQL's `FOR UPDATE SKIP LOCKED` can let queue consumers skip rows another transaction has locked. Its documentation explicitly notes queue-like use cases and warns that this is not a generally consistent view of data. Use it for the claim operation, not as a substitute for every concurrency rule. [S07](#)

A simplified transaction looks like this:

```
BEGIN;
SELECT id
FROM jobs
WHERE project_id = $1
  AND pool_id = $2
  AND state = 'queued'
  AND ready_at <= statement_timestamp()
ORDER BY priority DESC, ready_at, id
FOR UPDATE SKIP LOCKED
LIMIT 1;
-- Update the selected job and insert its new attempt here.
COMMIT;
```

The parameters denote driver-bound values. This fragment explains selection; the complete illustrative CTE in `examples/sql/claim.sql` performs the update and attempt insertion together. It still assumes that the caller has authorized the project and pool and arranged any required capacity reservation.

Keep the lock scope small

Never keep the transaction open while a runner downloads an image or performs a build. A claim commits before execution begins. Store an expiry time and fence generation so later messages can prove they refer to the current claim. Heartbeats extend ownership through conditional updates, not by retaining a database connection for the duration of the job.

Locks need a consistent order when multiple resources are involved. For example, reserve an organization concurrency slot, then a pool slot, then a job according to a documented order. A separate `COUNT(*)` followed by an update is vulnerable to concurrent admission decisions. Use locked capacity records or another atomic reservation mechanism, and release reservations through the same terminal-state transaction.

Shared deployment targets need stronger treatment than ordinary build slots. A target reservation can remain held while execution is uncertain. Expiring a runner lease must not silently authorize a second deployment to the same target.

Define the database contract

Use constraints for invariants the database can express: unique attempt identity, unique generation per job, valid state values, immutable delivery receipt keys, and foreign-key ownership. Store enough attempt history to audit retries without overloading the current job row with every observation.

A practical index begins with fields used to narrow eligible work, such as pool and state, and includes the ordering fields. Measure representative queries with the actual workload. A partial index on queued rows may help, but it is a design to benchmark rather than a guaranteed improvement for every data distribution.

PostgreSQL's default Read Committed isolation gives statements their own snapshots. More stringent transaction isolation can require explicit retry handling. Choose isolation and locking together, and retry the whole intended transaction when its documented conflict condition permits it. [S18](#)

Use the database clock for ownership comparisons. Keep transaction duration short enough that the chosen timestamp semantics are intentional. Wall-clock adjustment still matters operationally: leases are a coordination policy, not proof that a disconnected process has stopped.

Explain queue delay

Persist or derive a reason for why a job is not currently claimable: dependency not satisfied, approval required, no authorized runner, insufficient capacity, paused project, delayed retry, or uncertain target. Do not update thousands of rows every second just to populate a dashboard; combine durable blockers with current pool summaries.

Fairness is a separate concern from atomicity. A priority order may starve low-priority projects. Introduce a deliberate policy such as bounded per-project admission or weighted rotation, and test it with asymmetric workloads. “The query is fast” is not evidence that the queue is fair.

Exercise

Two schedulers both observe that an organization has four running jobs and a limit of five. They each claim another job. The row locks on the jobs work correctly. Why is the limit still violated?

Worked answer

The schedulers locked different job rows but made their capacity decisions from separate observations. Serialize admission through an organization capacity record or an equivalent atomic reservation. Claiming a job and reserving its slot must succeed or roll back together. A later metrics correction cannot undo the fact that six jobs were admitted.

Completion evidence

Run live multi-client tests for claim uniqueness, capacity limits, deadlock retries, transaction aborts, and recovery after losing the connection immediately after commit. The book's SQL is illustrative until those tests are executed against PostgreSQL.

10. Fence attempts and preserve uncertainty

A timeout is not a stopped process

A runner starts a deployment and then loses its network connection. The server's lease expires. The deployment may still be running perfectly well on the target. Starting another attempt immediately can run two releases concurrently, even if the database believes there is only one owner.

Distinguish **control ownership** from **physical execution**. A lease gives a runner time-limited permission to report and redeem capabilities. A monotonically increasing fence identifies the generation of that permission. Neither can reach backward through a network partition to terminate an arbitrary external command.

Every current-attempt message should identify the job, attempt, runner, and fence. Result acceptance is a conditional transaction that checks all four, the current state, and the lease policy. A zero-row update means the claim is no longer current; it is not a reason to retry the same update without predicates.

Walk through a stale completion

At time t_0 , runner R1 receives attempt A1 with fence 7. At t_1 , ownership expires and the job becomes uncertain. An authorized reconciliation establishes that the old execution stopped. A retry creates A2 with fence 8. At t_2 , R1 reconnects and reports success for A1.

The platform retains the report as a late observation if useful, but it must not replace A2's current state. The laboratory's `TestStaleCompletion` exercises this distinction. Its old attempt remains in the ledger with its own result, and the new attempt has a larger generation.

```
job J
  attempt A1 / fence 7 / observed stopped / failed
  attempt A2 / fence 8 / current / running
late report for A1 -> rejected as current completion
```

A fence must be scoped so it cannot be confused across jobs or projects. A numeric generation alone is insufficient. Bind it to a specific attempt and runner identity, and authorize the owning project separately.

Heartbeats must not resurrect old authority

A heartbeat arriving after lease expiry should not blindly extend the row. Require the same current claim, an active state, and an unexpired lease under the chosen policy. Rejoining after a longer interruption is a reconciliation flow, not a heartbeat with a generous timeout.

Also distinguish the result path from duplicate acknowledgement. If a valid terminal result was already committed, an identical retry may be acknowledged after the lease time has passed. It does not renew execution rights. A conflicting terminal result must be rejected or recorded as a conflict without replacing the committed outcome.

Runner clock time is diagnostic data, not authority for lease validity. Record the server's receipt time and the worker's observation time separately. This helps operators see whether a report is delayed without allowing a worker to extend its own lease by reporting a favorable clock value.

Fence external targets where possible

The control database can reject stale results, but an external target may need its own defense. Prefer idempotent operations keyed by deployment identity, compare-and-set target revisions, or an adapter that checks a target-side generation before mutation. These controls must actually be implemented at the target boundary; adding a fence field to an HTTP payload does not help if the receiver ignores it.

For a target that cannot enforce such a contract, use a conservative uncertainty policy: retain the lock, inspect the target, and require a controlled recovery decision. Document the residual risk. A generic shell deployment cannot honestly promise exactly-once effects.

Exercise

A scheduler stores fence 11, while the target still accepts commands from a disconnected worker carrying fence 10. The worker cannot update the CI database. Is the deployment safe?

Worked answer

The database is protected from stale state updates, but the target is not protected from stale side effects. The target must enforce an idempotency or generation rule, or the system must prevent another deployment until the old activity is reconciled. Fencing is only as strong as the boundary that checks it.

Completion evidence

Inject delayed heartbeats, duplicate results, conflicting results, old-runner reconnects, clock skew, and disconnects around target mutation. The expected state must name uncertainty explicitly rather than silently returning work to the queue.

11. Design an outbound runner protocol

Let runners ask for work

The runner initiates HTTPS connections to the server. This avoids requiring a publicly reachable listener on each worker host. It is an operational choice, not an authorization shortcut. The runner still validates the server identity, and the server still restricts what that runner may receive or submit.

A minimal protocol needs enrollment, capability advertisement, claim acquisition, start acknowledgement, heartbeat, log upload, result submission, cancellation observation, and reconciliation. Version these messages before adding optimizations. Start with bounded long polling if it meets the latency target; a streaming transport does not remove the need for durable request identities.

A claim reply should contain an immutable plan fragment, current attempt identity, fence, lease duration, and the exact resource references the runner may use. It should not contain broad administrative credentials or arbitrary instructions to read host files. Prefer scoped handles that can be redeemed only for the active attempt.

Negotiate capabilities without trusting labels

A runner advertises architecture, operating system, executor features, and configured capacity. The server matches required capabilities only within an authorized pool. A runner saying it supports privileged execution does not give a project permission to use that mode.

Store the runner version and protocol range. Reject incompatible workers before they claim work. During a rolling upgrade, keep a tested overlap range or drain old workers. A mixed fleet is normal; an undocumented mixed fleet is an incident waiting for a rarely used message type.

The runner's capacity is a promise about its own local admission. The server's capacity reservations are a promise about the global policy. Both need reconciliation after a restart. If the runtime shows two managed containers but the runner's local state file is empty, starting two additional jobs because the configured concurrency is two is wrong.

Make every mutation replayable

If a claim reply is lost, the runner cannot know whether the server reserved work. Give claim requests an idempotency identity or expose a “current assignments for this runner” recovery endpoint. Otherwise, a retry can strand one assignment and obtain another.

For logs and results, repeat the same attempt and segment identifiers. For credential redemption, bind the request to an authorized capability rather than accepting a free-form secret name. For cancellation, report whether the runtime observed termination; do not merely acknowledge that a cancel message was received.

Use bounded exponential backoff with jitter on connection failures. Keep heartbeats distinct from high-volume log transfer so a slow object upload does not accidentally expire execution ownership. All buffers and retry queues need limits and visible error behavior.

Drain and restart deliberately

Draining a runner stops new claims while allowing current attempts to finish or enter controlled cancellation. A runner upgrade should drain, persist its assignment journal, inspect the runtime, restart, and reconcile before admitting new work. Record the last confirmed log sequence and result state for each assignment.

The journal is not the authoritative job database, but it is evidence about local processes. Persist it atomically with safe file permissions. Never store plaintext long-lived secrets in a convenient diagnostic bundle. Keep credentials in a separate protected location and redact the bundle by construction.

A forced shutdown may leave physical work running. On return, the runner identifies its containers using explicit ownership labels and known attempt identifiers. It must not delete every container on a shared host to restore a clean-looking state.

Exercise

The server claims a job and commits, but the HTTP reply is lost. The runner retries without a stable request identifier. What could go wrong, and what does the protocol need?

Worked answer

The first assignment can remain leased without the runner knowing its identity, while the retry claims a second job. Use a stable claim request key with a stored response or a reconciliation endpoint that returns the runner's current assignments. The runner should recover the first assignment or explicitly abandon it through the contract before treating a new response as unrelated work.

Completion evidence

Run protocol tests with a proxy that drops replies after the server commits, interrupts uploads, duplicates messages, and disconnects during shutdown. Test the supported version matrix using actual built runner and server binaries.

12. Execute containers within explicit limits

Define what the executor owns

The executor turns a compiled job into runtime operations. It owns workspace creation, checkout material, container lifecycle, resource limits, output capture, exit observation, and cleanup. It does not own project authorization or decide that a production credential is safe to release.

For the initial design, use dedicated Linux runner hosts with a container executor. Keep the control plane elsewhere. Docker documents important limits and the authority of its daemon; a container is not a complete hostile-tenant boundary merely because it has a separate filesystem view. [S08](#)

Job containers should not receive the host daemon socket, unrestricted host mounts, host networking, or privileged mode by default. Place the daemon control connection in the runner's authority boundary. Stronger public multi-tenant execution is separate work, not a configuration checkbox on this trusted-runner design.

Build an execution envelope

The envelope includes image identity, working directory, command representation, environment, allowed mounts, network policy, CPU and memory budget, process limit, timeout, and output quota. Validate it against server and runner policy. A field absent from the plan must not silently inherit a permissive host default.

Resolve image aliases to immutable identities before execution and validate the allowed source registry. An image tag alone can change between runs. Store the selected platform when the image supports multiple architectures. Report pull and creation failures separately from a test command's nonzero exit.

Prefer explicit argument arrays for built-in actions. A shell step is a deliberate language boundary: document the shell, flags, working directory, and failure semantics. Do not concatenate a pull-request title or user parameter into shell syntax. Treat those values as data passed through a safe argument or environment mechanism, while still restricting which programs are allowed in privileged adapters.

Observe exit before declaring completion

A timeout produces a termination request. Apply a configured grace period, then escalate within the runtime's supported operations if necessary. Observe that the managed container stopped. Record whether the outcome was command failure, timeout, cancellation, infrastructure loss, or uncertain termination.

A successful signal operation is not necessarily a successful cleanup. Mounted volumes, child containers created through unauthorized daemon access, or target-side commands can outlive the main process. The executor contract must explain which resources it controls and which observations it can actually make.

Use a new workspace per attempt. Different jobs exchange artifacts explicitly rather than relying on a previously used host directory. Within a job, step workspace sharing should be intentional and documented. Clear checkout credentials before user-controlled steps start whenever the fetch mechanism permits it.

Cleanup must be conservative and repeatable

Assign ownership metadata at resource creation. Cleanup queries should filter by platform identity, runner identity, and attempt identity. Validate paths before recursive deletion, reject symlink escapes, and confine cleanup roots to configured directories. A faulty cleanup routine can be more destructive than the failed build it is trying to tidy up.

On disk pressure, stop admitting new work before exhausting the runner journal or control process. Report which budget is full. Artifact, cache, image, and workspace retention have different policies; deleting all four indiscriminately loses useful evidence and often hides the real capacity problem.

The executor should be testable behind an interface, but runtime integration tests remain mandatory. A mocked "container stopped" response does not establish that the selected runtime kills descendants or enforces a memory limit on the intended host configuration.

Exercise

A runner times out a job, sends a termination signal, and immediately marks it cancelled. Its cleanup call fails because the runtime is unavailable. What status should be visible?

Worked answer

Show that cancellation was requested and that physical termination is not confirmed. Preserve the attempt and quarantine affected capacity until reconciliation establishes what is still running. Do not release a deployment target lock based only on a signal request. A later observation can establish a terminal outcome while preserving the earlier uncertainty interval.

Completion evidence

Test CPU, memory, process, disk, timeout, mount, and output limits against the actual runtime. Test restart recovery and ensure cleanup cannot target unrelated containers or paths.

13. Move live logs without hiding gaps

Logs are streams with identity

A useful log protocol identifies organization, project, run, attempt, stream, and segment sequence. The attempt matters: a retry must not append into the previous attempt's output as if the two were one execution. Keep stdout and stderr either separately identified or explicitly framed in a combined stream.

Store bounded segments in object storage and their ownership, ordering, hash, size, and retention metadata in PostgreSQL. Do not insert one database row per arbitrary log line. The runner uses a bounded local spool and resumes from the last acknowledged sequence after reconnecting.

The laboratory's LogStore models a single stream. It accepts the next sequence, acknowledges identical retries, rejects conflicting bytes for an existing sequence, and refuses gaps. It copies input data rather than trusting the caller not to mutate a buffer later. Its memory quota is a teaching constraint; durable object storage needs a separate transaction and garbage-collection design.

Commit metadata after durable bytes

An upload normally writes bytes to an attempt-scoped temporary object, verifies their hash and size, then finalizes the segment's metadata. Do not make metadata claim that a segment exists before the storage operation has met the selected durability contract. A failed metadata commit may leave an orphan object; collect it after a retention grace period rather than exposing it as accepted output.

A signed upload URL is a capability. Limit it to one scoped key, method, size policy, and short lifetime where the storage service supports those constraints. Validate ownership when finalizing. An old attempt must not obtain a path belonging to a newer attempt or another project.

When the spool fills, choose an explicit policy: block output consumption, terminate the job, or record an intentional truncation with a visible gap. Silent dropping is not acceptable. Backpressure itself can change process behavior, so the chosen policy belongs in the executor contract and tests.

Separate transport from browser state

Server-Sent Events provides an event stream with identifiers and reconnection behavior, including a last-event identifier. It is a useful browser transport, but replay is still the application's responsibility. The stream format and browser behavior are specified by the HTML standard. [S14](#)

Send resumable cursors tied to the authorized stream. A browser reconnecting with an old cursor should receive the missing range or an explicit retention-expired response. Do not invent continuity when the requested logs have been deleted. Authorize the reconnect just as carefully as the first request.

Use a same-origin session strategy or a carefully scoped alternative for browser authentication. Avoid embedding durable bearer tokens into query strings that may appear in logs and history. Configure proxy buffering and idle timeouts intentionally, and test the deployed proxy path rather than only a direct development connection.

Render repository-controlled output as data

Log text can contain HTML, terminal control sequences, misleading links, and enormous lines. Escape it, restrict any supported ANSI formatting, and provide a plain-text view. A line saying “click here to authorize deployment” is not part of the platform interface simply because a build printed it.

The browser should virtualize or page large outputs instead of holding every line in the DOM. Keep tail-following optional so the viewport does not jump while an operator reads an earlier error. Display truncation, missing segments, reconnecting state, and retention boundaries visibly.

Secret redaction must account for secrets split across transport chunks, but it is still a secondary defense. Never treat a redaction helper as permission to expose credentials to code that should not have them.

Exercise

A runner retries segment 12 after a lost acknowledgement. The stored segment 12 has a different hash. Should the server overwrite it because the retry is more recent?

Worked answer

No. The same segment identity must denote immutable bytes. Reject the conflict and preserve the original accepted segment. A mutable overwrite could corrupt evidence or let a stale process rewrite history. The runner should diagnose its spool or identity mismatch rather than assign new meaning to an existing sequence.

Completion evidence

Test lost acknowledgements, duplicate segments, conflicting content, quota exhaustion, reconnect cursors, expired retention, browser escaping, and an object write that succeeds before metadata commit fails.

14. Preserve artifacts and keep cache trust separate

Artifacts are outputs; caches are hints

An artifact is an identified output that another job or operator may rely on. A cache is an optimization whose absence must not invalidate correctness. Treating the two as interchangeable makes releases depend on mutable, evictable data.

An artifact manifest should record its producer attempt, project, logical name, content digest, size, content type, creation time, and retention policy. Finalize it before downstream jobs reference it. Download authorization checks ownership and current access; object-key obscurity is not access control.

A cache entry needs a namespace, key, trust level, producer information, size limit, and eviction policy. A pull-request job should not be able to overwrite a cache later consumed as trusted release input. Separate trust domains even when their dependency lockfiles happen to match.

Transfer files deliberately

Jobs do not share a filesystem automatically. A producer declares the files it exports, and a consumer declares the artifact identity it imports. This makes it possible to schedule jobs on different hosts and understand which bytes crossed the boundary.

Before uploading, confine paths to the workspace, reject unwanted symlinks, and apply file-count and byte limits. Before extracting archives, reject absolute paths, parent traversal, unsafe links, device files, and excessive expansion. Validate the destination after path normalization, not just the original filename string. A compressed upload limit alone does not bound the uncompressed result.

Content hashes help detect corruption and identify immutable bytes, but they do not prove that the producer was authorized or the source was trustworthy. Keep the identity and policy record alongside the hash. A malicious build can produce a perfectly valid digest of malicious content.

Reports require their own parser policy

JUnit-style XML can improve failure summaries, but report files are controlled by the job. Use a parser configuration that does not resolve external entities, and enforce depth, element count, string length, and total-size limits. Normalize paths before linking report entries back to repository files.

Do not let a malformed report erase the actual command result. Record report ingestion as a separate result: accepted, partially accepted under a documented policy, or rejected. The operator should still be able to inspect the underlying authorized artifact or log when appropriate.

Test names and failure messages are untrusted display text. Escape them in the web interface. A report containing a test named “Deployment approved by administrator” must not resemble a real approval event.

Retention is part of the product

Define retention for ordinary runs, successful release artifacts, failed-run diagnostics, and audit records separately. A release still deployed to production may need its artifact retained even if the original build's default retention period has passed.

Garbage collection should first mark eligibility and then delete in bounded batches. Race it against new references in tests. A deletion record makes it possible to distinguish an intentional retention action from unexplained storage loss. Signed URLs should expire even when the underlying object remains.

Capacity controls belong to projects or organizations as well as the storage service. Otherwise one large test suite can exhaust a shared bucket or runner disk. Show current usage and the effect of retention policy changes before applying destructive operations.

Exercise

A project uses one cache key, `node-modules`, for both public pull requests and production builds. A release build is fast and its tests pass. What additional property is missing from the evidence?

Worked answer

The evidence does not establish that release inputs came from an authorized trust domain. The shared writable cache can influence a later trusted build. Use trust-aware namespaces and restricted writers, include relevant dependency and platform identities in keys, and ensure correctness can be reproduced from declared sources with an empty cache. Passing tests do not repair an undefined supply-chain boundary.

Completion evidence

Test cross-project download denial, stale upload capabilities, archive traversal and expansion limits, report parser limits, cache poisoning boundaries, and retention racing a newly created release reference.

15. Build once and preserve the image digest

Reuse a builder rather than inventing one

Modern CI orchestrates image builds through BuildKit. BuildKit already provides a build graph, caching, and image-output machinery; the platform's responsibility is to constrain inputs, assign authority, record outputs, and surface failures. The upstream project documents its execution and cache capabilities. [S09](#)

Keep the builder's authority separate from the job container. A job does not need a host Docker socket simply to request a controlled image build. Select an executor arrangement suitable for the trusted-runner boundary and document what the builder can access on that host.

Image construction has phases worth reporting separately: context preparation, source resolution, build execution, export, and optional registry publication. A successful build with a failed registry push is not a published release. Capture the final digest only from a confirmed output operation.

Make build inputs identifiable

A build record should name source commit, Dockerfile digest, context digest or a defensible context manifest, platform, builder version, declared build parameters, and selected base-image identities where resolved. This does not magically make arbitrary network-dependent Dockerfiles reproducible. It identifies what the platform can observe and where uncontrolled inputs remain.

Each application repository owns its Dockerfile and complete build context. A server image build from `apps/server` must not silently copy sibling source from the workspace. Resolve versioned modules or prepare a verified vendor snapshot before the build. A convenient root context can conceal that an application is not independently releasable.

Exclude credentials, local configuration, `.git` content when unnecessary, and oversized directories with an appropriate `.dockerignore`. Review what actually enters the context rather than relying only on intent.

Distinguish registry cache from artifact storage

Docker documents multiple cache backends, including a registry backend. The availability of a particular backend depends on the builder configuration. S3-compatible storage for platform artifacts does not mean the same deployment automatically supports every BuildKit cache mode. [S10](#)

Begin with an explicitly configured registry cache under project and trust-scoped references. Readers and writers should have different grants where appropriate. Cache failures can degrade performance without turning a successful build into an unverifiable output, provided the build remains correct without the cache.

Do not inject secrets through build arguments or permanent environment layers. Docker's build-secret mechanism provides temporary secret mounts intended for the build operation. The Dockerfile and builder still decide how that secret is used; granting it to untrusted build logic remains unsafe. [S11](#)

Promote the artifact, not the source branch

Once a tested image digest exists, deployment should reference that digest. Rebuilding the same commit for production can produce different bytes if external dependencies, tags, timestamps, or network results change. Build once and promote the confirmed artifact through environments, recording each approval and target observation.

For a multi-platform image, record whether the release identity is an image index or a platform-specific manifest. The target adapter must verify that the selected runtime platform is supported. Avoid using a local image ID as though it were the published registry digest.

A release manifest can include provenance information, but provenance is evidence about a build process, not a universal statement that the output is safe. Tie it to the builder identity and verification policy, then retain the corresponding source and test records.

Exercise

Staging runs `application:latest`. Production rebuilds the same commit and also runs `application:latest`. Both deployments succeed. Has the team demonstrated that production uses the bytes tested in staging?

Worked answer

No. The tag is mutable, and a second build may resolve different inputs. Record the finalized registry digest from the first build and promote that exact identity. Verify the target's observed digest after rollout. Source equality and tag equality are weaker than artifact identity.

Completion evidence

Test independent build contexts, secret handling, cache isolation, export versus push failure, architecture mismatch, digest capture, and promotion without rebuilding. Do not claim registry publication was tested unless it actually occurred in an authorized test registry.

16. Integrate source events, schedules, and matrices

Keep trigger intent explicit

A run may be requested manually, by a push, by a pull-request event, by a schedule, or by another controlled automation. Record trigger kind, source identity, requester, event identity, and the policy decision. These fields help explain why a run exists and which restrictions applied.

A GitHub App integration should request only the permissions needed for repository discovery, source access, and status reporting. Keep installation identity separate from a human user's login identity. Revoking a user's access and uninstalling the integration are different lifecycle events. Handle both rather than treating the first successful OAuth or installation flow as permanent authorization.

Deduplicate provider deliveries using their delivery identity, as Chapter 8 describes. A manual rebuild of the same source commit is still a distinct intent when it uses a new idempotency key. Do not collapse the two through a global “commit already built” flag.

Pull requests are a policy boundary

The same project may contain trusted branch builds and less-trusted contribution builds. Decide which source revision is tested: the contributor head, a provider-created merge revision, or a locally constructed merge. Record both the requested references and the actual checked-out commit. This is essential when a branch changes while a run waits in the queue.

Do not combine privileged workflow context with untrusted checked-out code. GitHub's secure-use reference discusses the hazards of privileged triggers and self-hosted execution. In Modern CI, the server must enforce the permitted pool, credential set, and cache namespace independently of the contributor's YAML.

[S17](#)

A status check belongs to a specific run and commit. Give it a stable external identity for update retries. When an older run finishes late, update its own check rather than whichever check is currently most recent for the branch.

Schedule occurrences, not timer callbacks

A scheduler process can restart, overlap with another instance, or miss a timer while the database is unavailable. Persist each schedule's timezone, expression, next intended occurrence, and catch-up policy. Use a unique occurrence identity such as schedule ID plus the scheduled UTC instant.

When daylight-saving rules produce a repeated local time, the design must specify whether both instants run. When a local time does not exist, it must specify whether the occurrence is skipped or moved. Do not let the behavior depend accidentally on which host happened to run the callback.

On recovery, apply a bounded catch-up rule. A monthly administrative job may need one missed occurrence; a job scheduled every minute should not necessarily flood the queue with thousands of runs after a long outage. Make the policy visible to operators and preserve skipped occurrences as meaningful scheduling decisions where required.

Expand matrices before dispatch

A matrix creates multiple concrete jobs from declared dimensions. Calculate its maximum expansion before allocation and reject excessive size. Include each dimension combination in the compiled identity so its logs and artifacts cannot collide with another member.

Define aggregate success carefully. A cancelled member, an allowed-to-fail member, and a member that never started are not necessarily equivalent. A fail-fast policy may request cancellation of remaining members, but their physical termination still follows the normal runner protocol.

Reusable templates should be resolved to immutable versions when compiling the run. Avoid a template branch that changes silently between two jobs in the same run. Store a receipt of template source and resolved inputs alongside the frozen plan.

Exercise

Two scheduler instances wake after an outage and each calculates that the 09:00 occurrence was missed. What durable key prevents duplicate scheduled runs, and why is a process-local mutex insufficient?

Worked answer

Use a unique database key based on schedule identity and the intended occurrence instant, within the transaction that creates or enqueues the run. A process-local mutex coordinates only threads in one process and vanishes on restart. Both instances can legitimately calculate the same missed occurrence; only the durable uniqueness rule resolves which one creates it.

Completion evidence

Test duplicate event delivery, changed branch heads, revoked installations, restricted pull-request execution, repeated and nonexistent local times, bounded catch-up, matrix limits, and out-of-order status updates.

17. Give credentials to an authorized attempt, not a script name

A secret reference is not a secret value

The pipeline snapshot stores a reference to a credential and the policy under which it may be used. It must not store the plaintext secret. The server resolves that reference only for an authorized active attempt and only within the permitted project, environment, and runner pool.

Make secret creation write-only through normal APIs. Return metadata such as name, scope, version, and last rotation time, not the value. A “show value” endpoint creates another sensitive access path and should not appear accidentally as part of a generic resource serializer.

Encrypt stored values with an established cryptographic library and an operator-managed key hierarchy. Store key version metadata so rotation and restore are possible. Keep encryption keys outside the database backup they protect. Losing the key while successfully restoring every database row is still losing the secret service.

Credential redemption is a policy check

A runner presents its current attempt identity, fence, and the scoped capability named in its plan. The server verifies that the attempt is active, the pool is authorized, the project still has access, and the target policy has not been revoked. Only then does it return the permitted material or a short-lived external credential.

An expired lease must prevent new redemption. Previously delivered secrets cannot be retrieved from a worker's memory by revoking a database row. Previously issued external tokens can remain valid until their expiry or provider-side revocation. Design short lifetimes and narrow scopes around that reality rather than claiming immediate universal revocation.

Separate checkout credentials from deployment credentials. A build that can fetch one repository should not automatically be able to push to it or access every repository installed under an organization integration. Remove temporary fetch material before running contributor-controlled steps where possible.

Use federation deliberately

OpenID Connect can let a workload exchange a signed identity assertion for short-lived cloud credentials instead of storing a long-lived cloud key. The provider trust policy must validate issuer, audience, and a suitably narrow subject. GitHub's OIDC documentation explains this federation pattern. [S19](#)

For a Modern CI implementation, start disabled. Enable it only after configuring the issuer, key distribution, trusted audience, subject claims, and provider policy. Bind claims to the project, environment, and approved workload identity. Do not let YAML choose an arbitrary role identifier and assume the provider will enforce your application's intention.

Use a maintained token library. Validate time bounds, accepted algorithms, key identity, and claim types. Keep signing-key rotation and stale-key caching in the runbook. A local mock token exchange is useful for contract tests but does not prove that the actual provider policy is correctly configured.

Redaction is a secondary defense

Redact credentials from structured logs, error messages, traces, and support bundles. For streamed output, consider values split across chunks and encoded in common forms. But do not promise that masking prevents deliberate exfiltration. Code that receives a secret can transform or use it in ways a text filter cannot safely recognize.

The stronger control is deciding which code receives which authority. A deployment adapter that accepts an immutable release subject and a registered target can be narrower than giving a generic shell a cloud administrator token. The adapter still requires review and testing; a friendly function name does not establish least privilege.

Audit secret policy changes and redemptions with metadata only. Useful fields include project, attempt, credential version, requesting runner, policy decision, and expiry. An audit entry containing the secret defeats the purpose of encrypting the main table.

Exercise

A secret is revoked at 14:00. A worker obtained an external token at 13:59 that expires at 14:09. The interface displays “all access removed.” What should it display instead?

Worked answer

It should distinguish blocked future redemption from already issued authority. Show that the secret or policy is revoked for new requests, and record the latest known expiry of issued credentials where the platform can track it. Provider-side revocation may shorten that interval if supported, but the system must not claim it occurred without confirmation.

Completion evidence

Test revoked leases, cross-project redemption, key rotation, backup without keys, restored keys, token audience and subject restrictions, expired assertions, and accidental leakage through diagnostics.

18. Deploy an approved immutable release

An environment is a policy-bearing resource

A production environment is not just a string in YAML. It has an owner, authorized projects, approved target adapters, allowed credential references, concurrency rules, approval policy, and an audit history. The server enforces these rules independently of the pipeline author's requested configuration.

A deployment request binds a release artifact to an environment and target configuration. The artifact must be finalized and immutable. Record a configuration digest alongside the image digest, because changing environment settings can materially change the deployment even when the image stays the same.

Do not allow a job author to remove an obligatory production approval by deleting a YAML field. The compiled plan can request deployment, but the environment policy decides whether dispatch is allowed.

Approve a subject, not a label

An approval should identify project, environment, artifact digest, configuration digest, and policy revision. It should include approver identity, time, expiry, and any required separation-of-duties rule. “Approve release latest” is not a stable authorization.

The included approval helper compares an exact subject and refuses expired or revoked approvals. It does not implement membership or approver-role checks; the product must evaluate those at the service boundary and recheck relevant revocations before dispatch.

If the artifact, target configuration, or approval policy changes, the old approval no longer matches. Ask for a new decision rather than silently attaching the old human action to a different operation. The interface should make the changed field visible so reapproval is meaningful.

Serialize access to the target

Acquire an environment or target reservation before mutation. Keep its identity in the deployment record and executor plan. A second deployment to the same target must wait unless the adapter's concurrency model explicitly supports overlap.

An SSH adapter should accept a registered target with a pinned host identity, restricted account, known commands or release procedure, and a specific artifact. Do not disable host-key verification to make onboarding easy. A user-supplied host and shell fragment create a much broader product than a controlled deployment adapter.

Stage the release, verify expected inputs, perform the documented transition, and probe health through a target-specific contract. Record what was observed, not simply that the SSH process exited with zero. Some services launch asynchronous rollout work and return before the new version is healthy.

Observe the deployed identity

A deployment can be requested, approved, dispatched, applying, verifying, healthy, failed, or uncertain. Keep these states separate from the upstream build's status. A successful image build does not mean that the application is running in production.

The target observation should identify the actual deployed digest or release revision where the platform can inspect it. Store health-check parameters and results. A generic 200 OK from an unrelated route is weak evidence if it cannot establish which version answered.

Notifications come after committed state transitions through the outbox. Include environment, release identity, and a link to the deployment record. Avoid messages whose wording suggests that receiving a notification is itself proof of a healthy rollout.

Exercise

An approver authorizes digest D1. A developer changes the deployment configuration and reuses the approval because the image digest is unchanged. Is that valid under this design?

Worked answer

No. The configuration digest is part of the subject. Changing a database endpoint, feature flag, volume, or runtime command can change the operation substantially. Compile a new subject and obtain the required approval. The old decision remains valid only for the exact operation it named.

Completion evidence

Test artifact substitution, configuration changes, revoked approver access, policy changes, concurrent deployment requests, host-key mismatch, asynchronous rollout failure, and verification that reports the wrong deployed identity.

19. Recover from cancellation, uncertainty, and rollback

Cancellation has a timeline

A queued job can often be cancelled without contacting a runner. An active job requires a request, delivery, runtime action, and a stopped observation. A deployment may have already changed the target before any of those steps finish. Keep the cancellation request timestamp and the observed terminal timestamp distinct.

If cancellation races a successful exit, preserve the actual observation. A request to stop does not prove that the operation was stopped. The laboratory permits a success result arriving after a cancellation request while the same claim is still active. The user can see that cancellation was requested too late rather than being shown a fictional cancelled deployment.

A timeout follows the same principle. It is a policy deadline for permitted execution, not a guarantee that every external side effect has stopped at that instant.

Quarantine uncertain targets

When a runner disappears during a target mutation, mark the deployment uncertain and retain its target reservation. The recovery procedure should first establish runner state, then target state, then the relation between the two. An operator needs a stable deployment identity and a checklist, not a generic Retry button.

A useful reconciliation record includes the observer, time, target identity, observed revision, active operation identifier if available, and the resulting decision. A recovered connection can provide evidence, but it does not automatically make the old runner authoritative again. Its original attempt and fence still matter.

Permit read-only inspection with narrower credentials when possible. If the target can report a deployment operation by idempotency key, use that to distinguish “not applied,” “still applying,” “applied,” and “unknown.” An adapter unable to inspect state should declare that limitation.

Retry policies depend on side effects

A pure compilation step can often be retried after infrastructure failure. A package publication, payment-like external operation, or database migration may not be safe to repeat without additional identity checks. Classify retry safety explicitly rather than assigning the same `max_retries` setting to every job type.

Backoff is useful for transient faults but does not fix a semantic error. Retrying a bad credential, invalid pipeline, or unauthorized target just creates noise.

Record the reason for each retry and show attempts separately.

The platform should never advertise exactly-once execution for arbitrary shell commands. It can offer durable attempt identity, conditional state updates, idempotency contracts for specific adapters, and conservative reconciliation when the external boundary cannot provide certainty.

Rollback is another controlled change

Rollback normally selects a previously retained artifact and applies an adapter-specific reverse transition. It needs its own authorization, target reservation, and health verification. It is not a database time machine and cannot generally reverse arbitrary external effects.

A database schema change may prevent an old application binary from running. A data migration may have discarded information. A new external message may already have been consumed. Record rollback compatibility before release and distinguish an application rollback from a database restore.

When rollback itself becomes uncertain, do not erase the original deployment record. Link the operations and reconcile the actual target state. Operators need to see the complete sequence of intended and observed changes.

Exercise

A deploy command uploads a new image, runs a destructive data migration, and restarts the service. The health check fails. An automated rollback simply starts the previous image. What prerequisite was missing?

Worked answer

The release lacked a verified compatibility and recovery plan for the data change. The previous binary may no longer understand the schema or data. Use expansion-and-contraction changes where possible, test rollback within its stated window, and keep a separately tested restore procedure for changes that cannot be reversed safely. A previous image digest alone is not a complete rollback plan.

Completion evidence

Run failure experiments before and after each externally visible mutation. Demonstrate that target locks remain held through uncertainty and that recovery decisions preserve the original event history.

20. Build an operator interface that explains state

The dashboard is a diagnostic tool

A useful dashboard answers why work is waiting, what is currently executing, what changed, and what needs an operator decision. It should not only rank recent failures. The most valuable first screen may be a clear distinction between no eligible runner, exhausted capacity, missing approval, and uncertain deployment state.

Use Angular as a client of the versioned server API, not as an alternate policy engine. Organize features around projects, runs, runner pools, artifacts, environments, and administration. Keep generated contract types separate from presentation models and record the schema receipt described in Chapter 4.

Angular Signals provides reactive state primitives; the framework documentation explains how values and derived state are tracked. Use them for view state where appropriate, without mixing transport replay rules into arbitrary component side effects. [S12](#)

Snapshot first, ordered events second

Load an authorized snapshot with a revision or cursor, then subscribe to changes after that point. Without a consistent boundary, an event can arrive between the initial GET and the stream connection and disappear from the screen.

Apply events only when their identity and ordering match the current subscription. Duplicate events should be harmless. If the browser detects a gap or an unsupported event version, refresh the snapshot rather than guessing what state was missed. Switching projects must close the old subscription and clear sensitive view state.

The API remains authoritative. A Cancel button can optimistically display “requesting cancellation,” but it should not invent the terminal state before confirmation. An approval form should show the exact subject being approved and detect if it changed while the dialog was open.

Give large data a bounded presentation

Use pagination for history and server-side filters for large run sets. Use a virtualized or paged log view rather than adding every line to the DOM. Preserve selection, scroll position, and the user's decision to stop following live output.

The pipeline graph needs a keyboard-readable alternative, such as a job table with dependencies, state, and reason. Do not rely only on color to convey outcomes. A screen-reader label should distinguish failed, cancelled, blocked, and uncertain states instead of saying “red node.”

Timestamps should reveal the timezone and offer exact values. Show duration separately from wall-clock time. Distinguish queued time, execution time, and time spent awaiting approval so the product does not blame the runner for a human decision delay.

Treat output as untrusted text

Repository names, branch names, commit messages, report text, and logs can contain hostile markup. Angular's security guidance describes escaping, sanitization, and the risks of bypassing trust protections. Avoid inserting raw build output into HTML or using bypass APIs merely to preserve formatting.

S13

Use narrowly controlled rendering for any supported ANSI styles. External links should be visibly external, with an intentional navigation policy. Error messages returned by the server should not expose stack traces, SQL, credential material, or internal target URLs to an unauthorized viewer.

Access checks belong on every server endpoint. Client route guards improve navigation but do not establish security. Test direct API calls independently of what the interface chooses to display.

Design interruption as a normal state

A reconnecting banner should say whether data may be stale and when it was last confirmed. A deleted artifact should display the retention reason if available, not a generic unexpected error. An inaccessible project should clear cached content after access revocation.

When an action fails, preserve the operator's context. A failed deployment approval should not send the user back to a blank dashboard. Show the failure and the current subject so the next decision can be deliberate.

Exercise

The web page receives events 201 and 203 but never 202. Event 203 says the run succeeded. Should the client simply advance its cursor to 203?

Worked answer

Not unless the protocol explicitly makes event 203 a complete authoritative snapshot. With incremental events, the missing update may contain a critical state transition or resource change. Detect the gap and refresh from an authorized snapshot or replay endpoint. The browser must not manufacture a coherent history from incomplete data.

Completion evidence

Test event gaps, duplicate events, revocation during a session, escaped output, large logs, keyboard navigation, exact approval subjects, and reconnection through the actual production proxy path. This book contains the design, not a compiled Angular application.

21. Keep the CLI precise and extensions constrained

A CLI is another client, not an administration back door

The CLI should use the same versioned API and authorization decisions as the browser. Give it explicit contexts for server, organization, and project. Display the selected context before dangerous operations and avoid guessing an environment from a similar name.

Machine-readable output needs a stable contract. Send data to stdout and diagnostics to stderr. Use documented exit statuses to distinguish success, invalid input, authentication failure, authorization denial, remote failure, and an operation that remains uncertain. A successful HTTP submission is not necessarily a successful deployment.

A command such as `runs watch` should reconnect using the documented cursor and stop according to an explicit terminal-state rule. A command such as `deployments reconcile` should require a specific deployment identity and recorded evidence. Avoid broad wildcard mutation by default.

Make automation reviewable

Support a read-only plan or preview for changes that can be described before mutation. The preview should identify the actual resources and the version being changed. Use conditional updates to detect that a resource changed between preview and apply.

Interactive confirmation is useful for a human but not sufficient for scripted use. Provide an explicit noninteractive flag only when the request fully identifies its scope and desired action. Keep approval and authorization checks on the server even if the client has a `--yes` option.

Store credentials with restricted permissions. Do not accept a token in a command-line argument when a safer input path is available, because arguments may be visible in process listings or shell history. Support reading from a protected file or environment under a documented policy, and exclude secrets from debug output.

An extension contract is a permission contract

An action definition should declare its version, inputs, outputs, runtime identity, required capabilities, and supported protocol. Resolve it to an immutable identity during compilation. The manifest requests capabilities; it cannot grant them to itself.

Run extension code outside the control-plane process. For the initial product, a small built-in action set is easier to secure and support than a broad marketplace. Add a signed catalog or publisher-verification process only when there is a real need and a defined trust policy.

A containerized action still inherits whatever the executor gives it. Review mounts, credentials, network reachability, and output channels. Isolation boundaries come from the execution design, not from naming something a plugin.

Import Jenkins narrowly and honestly

The migration analyzer can inspect exported configurations and recognize a bounded subset of static stages, commands, credentials references, triggers, and artifact declarations. It should produce a translation report with supported, partially supported, and unsupported items.

Do not evaluate arbitrary Groovy while importing. Do not copy credential values into generated YAML. Map credential references and require an operator to establish the equivalent policy in the new system. Dynamic shared-library behavior may need manual redesign rather than a mechanical conversion.

Run the new pipeline in a non-production comparison mode before switching the authoritative deployment path. Compare source identity, outputs, test counts, and target behavior. Prevent both systems from deploying to production simultaneously during the transition.

Exercise

A migration tool can translate every shell command but drops an unsupported approval plugin. It reports “100% command coverage.” Is that enough to activate the imported pipeline?

Worked answer

No. The missing approval changes who may authorize a deployment. Migration coverage must include control and security semantics, not just executable commands. Mark the pipeline blocked until the environment policy and approval behavior are represented and tested. A mechanically complete command list can still be an unsafe migration.

Completion evidence

Test CLI exit statuses and output stability, stale previews, restricted credentials, version mismatch, action capability denial, and migration reports that never silently discard unsupported security controls.

22. Observe reliability, capacity, and recovery

Measure the waiting as well as the running

A platform can execute jobs quickly while users wait a long time for an eligible runner or approval. Measure request-to-queue latency, queue delay by reason, claim latency, runner start delay, execution duration, log lag, and result-commit delay separately.

Correlate logs and traces with run, job, attempt, runner, and deployment identifiers. Keep those identifiers out of unrestricted metric label sets when their cardinality would grow without bound. Metrics summarize behavior; detailed logs and traces explain a specific execution.

An uncertain-deployment count deserves more attention than a generic failure total. It represents work whose external effect is not understood. Alert on age and actionable state, with a runbook link and the relevant target identity, rather than paging on every transient retry.

Establish service objectives from use

Do not invent a public performance claim from a developer laptop. Start with proposed internal objectives, collect measurements, then adjust capacity and architecture. State the workload: graph size, concurrent jobs, log bytes per second, artifact sizes, runner distribution, and failure injection profile.

A useful objective might distinguish “authorized run accepted into durable storage” from “run started on an eligible runner.” The first measures control-plane responsiveness; the second includes capacity and policy delays. A single average hides which experience is deteriorating.

Load tests should include noisy projects and quiet projects together. Observe starvation, backlog growth, database contention, log storage pressure, and recovery after stopping the load. A system that survives a spike but never drains its backlog has not recovered.

Back up state and the ability to interpret it

PostgreSQL documents several backup approaches, including logical dumps and physical/continuous-archiving approaches. Choose a method around recovery-point and recovery-time objectives, then test restoration. A backup file's existence is not a restore result. [S20](#)

The recovery set also includes object storage, encryption keys, signing keys, integration configuration, source and release manifests, and the versions needed to interpret the database. Restoring rows without the keys needed to decrypt them is not a complete recovery.

Keep restored environments isolated from production side effects. Start the control plane with scheduling and dispatch paused. Reconcile old leases and external targets before enabling work. A restored database may remember an attempt as running even though the target changed after the backup point.

Plan a controlled restart

Document the restart sequence for API, scheduler, database, and runners. Decide which component is allowed to migrate schema, which checks compatibility, and which can safely resume dispatch. Ensure health endpoints distinguish process liveness from readiness to accept work.

A runner should stop admitting new jobs when its runtime, disk, credential source, or control connection no longer meets the required conditions. Do not advertise full capacity just because the agent process is alive.

For the first deployment, keep the infrastructure understandable. A control host with separate database and dedicated runner hosts may be easier to operate than a large service fleet. Add more control instances, broker infrastructure, or Kubernetes executors when measurements show a need and the failure model is ready.

Exercise

A database restore succeeds in a test environment. The scheduler immediately resumes jobs recorded as leased at the backup time. Why is this unsafe, and what is the first recovery action?

Worked answer

Those records may no longer match physical execution or target state. The original runners or deployments could have continued after the backup. Keep dispatch paused, establish a new controlled recovery context, and reconcile attempts and targets before issuing new authority. Restore validation must include side-effect control, not just the ability to run SQL queries.

Completion evidence

Record a real restore exercise with measured recovery times, key availability, object consistency checks, paused dispatch, lease reconciliation, and a controlled return to service. Publish performance numbers only with the workload and environment that produced them.

23. Test failures and upgrade through compatible states

Test the boundary that makes the claim

Unit tests should cover graph validity, state transitions, signature checks, approval identity, and size limits. Contract tests should exercise actual serialized requests and version combinations. Integration tests should use PostgreSQL, object storage, and the selected container runtime. End-to-end tests should follow a real repository through execution and controlled deployment.

Each layer answers a different question. The companion laboratory demonstrates a subset of unit-level semantics. Its race test does not validate PostgreSQL, and its approval helper does not establish a secure user interface. Keep the verification report explicit about those limits.

For stateful behavior, generate sequences of operations and assert invariants after every step: at most one current claim, monotonic fences, immutable terminal attempt results, no unauthorized credential redemption, no unlocked uncertain target, and no artifact reference to unfinalized bytes.

Put faults at the commit boundary

The most revealing failures happen just before and just after a state transition becomes durable. Drop the reply after a successful database commit. Kill the runner after starting a container but before writing its journal. Fail object metadata insertion after bytes were uploaded. Interrupt a deployment after the target applied the change but before the runner received confirmation.

These experiments should yield supported recovery states, not unexplained panics or duplicate side effects. Use disposable fixtures and registered test targets. Do not turn a failure test into an unsupervised production deployment.

Security tests need negative cases: cross-project identifiers, expired capabilities, forged runner identity, hostile archive paths, excessive matrix size, malicious report text, disallowed outbound destinations, and credential material in diagnostics. A successful authorized path does not imply that the denied path is correctly enforced.

Upgrade producers before removing old contracts

An additive protocol change can allow old and new runners to coexist temporarily. A schema expansion can let old and new server versions operate during a rollout. Plan these states explicitly and test them using built binaries, not only matching source from one workspace.

A migration should have a bounded operation strategy for large tables. Avoid locking a busy queue table for an uncontrolled duration during normal dispatch. Backfill separately when needed, track progress, and define an abort or resume path.

After the new representation is fully in use and the downgrade window is closed deliberately, a later release can remove the old field. Document that cutoff. A semantic change may be incompatible even when the field name and data type remain unchanged.

Make evidence refer to exact inputs

Record application commits, contracts revision, schema version, built image digests, test configuration, tool versions, start time, exit status, and output location. A green check attached to a different source snapshot is not evidence for the release being published.

Do not overwrite failed-run evidence with the next successful run. Keep a concise history so maintainers can see what changed and which failure was resolved. An operator must be able to distinguish “test passed,” “test skipped,” “not run,” and “blocked by missing infrastructure.”

The book's implementation ledger uses the same principle for work items: evidence is required before marking a task complete. It does not claim to execute or verify the entire product automatically.

Exercise

A compatibility test compiles the newest server and newest runner against the same local contracts checkout. The release plan allows a six-month-old runner to remain connected. What test is missing?

Worked answer

Run the new server against the actual supported older runner binary and exercise every relevant message and recovery flow. Also test rejection outside the supported range. Building both newest components from one checkout only proves a same-source combination, not the rolling-upgrade promise.

Completion evidence

Maintain an executable failure matrix and a supported-version matrix. Release gates fail when required tests are missing, not only when an executed test reports an error.

24. Publish a reproducible release and maintain the project

A release is more than a tag

A product release should name the workspace commit, all child source commits, built artifacts or image digests, contracts version, database schema range, supported runner versions, migration procedure, rollback constraints, and verification evidence. A source tag is useful, but it does not identify every runtime dependency or guarantee that a build will produce the same bytes.

Publish child dependencies before a workspace references them publicly. Verify that a fresh consumer can retrieve the pinned commits with the intended access level. A private test succeeding with the maintainer's credentials is not proof that an open-source reader can reproduce the checkout.

Keep release notes factual. State features actually implemented, known limitations, upgrade requirements, and security changes. Do not replace a missing benchmark with a slogan about speed. A reliable small scope is more useful than a long checklist whose entries have no executable evidence.

Migrate from Jenkins by proving equivalence where it matters

Inventory existing jobs, credentials, integrations, schedules, shared libraries, artifact retention, and deployment targets. Classify static supported configurations separately from dynamic or plugin-dependent behavior. An import report should identify gaps before any production switch.

Run a selected pipeline in parallel without giving both systems production mutation authority. Compare actual source revisions, test results, artifact identities, and notifications. Establish who owns the deployment schedule at each stage of migration. Keep a rollback path for the migration itself, including access to the previous job configuration and retained release artifacts.

Full Groovy or plugin compatibility is outside the first release. Document manual translations and the evidence used to approve them. Security policies are part of the migration, not something to add after the commands work.

Maintain an open project through reviewable changes

The book repository includes contributor guidance, issue templates, a pull-request template, a licensing map, and publication scripts. The application repositories should have comparable ownership rules and release responsibilities. Contributors need to know where a schema change belongs and how to update its consumers.

Require tests or a reproducible explanation for behavioral changes. Review new dependencies for maintenance, licensing, and privilege implications. Keep secrets out of issue reports; use a private reporting path for vulnerabilities when one has been configured. Do not promise an incident-response service level the maintainers cannot provide.

A public roadmap can contain eighty implementation work items without implying that they are complete. Mark scope and evidence clearly. When a task is blocked, record the missing prerequisite rather than changing its acceptance condition until it appears green.

Publish the book independently

The canonical manuscript is Markdown. The included builder exports a navigable static website, a print PDF, an EPUB, and a sample. The repository can be browsed directly without running the build. This keeps reading and contribution accessible even when the publication host changes.

GitHub Pages supports custom build workflows. The supplied workflow separates validation and artifact construction from the deployment job and limits deployment to the publication branch. Repository settings must enable GitHub Actions as the Pages source before deployment. A local site build is not a hosted deployment result. [S21](#) [S22](#)

Use a release tag for a reviewed book edition. Attach the PDF and EPUB to a release if desired, and keep the source history open. A reader should be able to identify which edition produced the download and submit a correction against the corresponding manuscript.

Know what remains beyond the first platform release

Kubernetes execution, autoscaling pools, hosted control-plane billing, public hostile-code execution, additional source providers, and broader operating-system support are separate future tracks. They reuse some contracts but change operational assumptions. Add them through scoped design and evidence, not by silently broadening the original security promise.

The durable core of the system is simpler to state: immutable intent, explicit authority, bounded execution, honest uncertainty, and observable recovery. Every new feature should preserve those properties.

Exercise

A workspace release is public, but its contracts tag exists only in a maintainer's local repository. The maintainer can build everything. What should the release gate do?

Worked answer

Fail the consumer-resolution gate. Publish the dependency through the authorized release process, ensure the pinned commit remains reachable, and repeat the independent build from a clean environment with the access level promised to readers. A local checkout cannot establish that someone else can reproduce the release.

Completion evidence

A fresh reader can obtain the source, understand the license, reproduce the documented checks, and distinguish implemented behavior from future work. A fresh operator can identify exact deployable outputs and follow a tested upgrade and recovery procedure.

Appendix A. Laboratory runbook

Scope and prerequisites

This runbook operates only the Go examples included with the book. It does not provision application repositories, a container daemon, a database, cloud resources, or a production deployment. Work in a disposable checkout. No credential is required by the Go examples, and they do not contact external services.

Install Git for version control, a supported Go compiler, and Python 3.11 or later for book tooling. The Go module declares a language baseline; use an actively maintained release when reproducing the exercises. The website and EPUB use the packages in `requirements-site.txt`. PDF export additionally uses `requirements-print.txt` and WeasyPrint's system dependencies. The exact preparation environment is recorded in `verification/`.

Inspect before running

From the repository root, inspect `examples/lab/go.mod` and the source files. The module contains no external dependencies and no general shell executor. Its demonstration runs the domain model only. Review the Makefile targets before using them. `make check` validates the publication sources and runs the example tests; `make site` and `make epub` write local publication outputs.

```
python3 -m venv .venv
. .venv/bin/activate
python -m pip install -r requirements-site.txt
make check
make site
make epub
```

The virtual environment is ignored by Git. The site is written to `dist/site/`. Serve that directory locally when testing navigation:

```
python3 -m http.server 8000 --directory dist/site \
--bind 127.0.0.1
```

This command binds to loopback. Open the local address printed by Python. Stop the server with the normal terminal interrupt when finished. Opening a single HTML file directly may work for navigation, but serving the directory more closely resembles a static host.

Read the execution model

Run the demonstration directly:

```
cd examples/lab
go run ./cmd/demo
```

The expected sequence is a compiled test, package, and deploy graph; an uncertain state after expiry; refusal of a blind retry; refusal of the old attempt's stale result; success for the authorized new attempt; and recognition of an identical log retry. If the output differs, run the tests and inspect the exact commit you are using before changing expected values.

The ledger contains no persistence. Re-running the demonstration creates a fresh in-memory model. This is intentional. Use the database exercises from Chapters 9 and 23 to validate a future durable implementation instead of assuming the model survives restart.

Reproduce the tests

```
cd examples/lab
go test -count=1 ./...
go test -race -count=1 ./...
go vet ./...
```

The first command disables cached success for that invocation. The race check requires a compatible compiler environment, including the platform support required by Go. A missing prerequisite should be recorded as a blocker, not relabeled as a successful check. The preparation report distinguishes tests actually run from target-system tests that remain unexecuted.

Inspect failures around the corresponding invariant. A cycle-test failure belongs to graph compilation. A stale-result failure belongs to identity and state acceptance. A signature test failing after JSON reformatting demonstrates why exact bytes matter. A log conflict is not repaired by overwriting the expected segment hash.

Explore safely

Make a local branch and introduce one small defect. Remove a fence comparison, allow a missing dependency, or make an approval ignore the configuration digest. The relevant test should fail. Restore the invariant and commit the repair with a short explanation. Do not weaken the test to make the defect appear acceptable.

For concurrency, run the single-claim test repeatedly. It verifies process-local admission under the ledger mutex. To extend the exercise to PostgreSQL, first build a disposable database harness around the SQL examples, then use multiple real connections and crash injection. That harness is not included as an already validated service.

SQL and runtime examples

`examples/sql/` contains a minimal schema and prepared statements explaining atomic claims and conditional completion. They are not complete production migrations. Tenant authorization, global capacity reservations, audit insertion, and external deployment safety require additional work. Do not run them against an existing application database.

Pipeline YAML files describe the proposed platform grammar. The book laboratory does not execute them. No example should be interpreted as a command to publish an image or deploy a service merely because it contains a deployment-related field.

Collect evidence and clean up

Save command output, tool version, exit status, source commit, and relevant configuration with an experiment. Remove credentials and unrelated local paths from reports before sharing them. The repository's preparation evidence is a record for this edition, not a replacement for your environment's results.

Delete `dist/` to remove local exports, or use `make clean`. The clean target does not delete the manuscript or the laboratory source. Delete the virtual environment only when you no longer need its installed dependencies. No cloud or production cleanup is required by the included Go demonstration.

Appendix B. From chapters to eighty implementation work items

The book presents the design in reading order. Product implementation has a different dependency order. The map below preserves eighty bounded work items for the eight-repository application workspace. These are specifications for future implementation, not claims that the companion examples complete the platform.

Every item starts as **planned** in `implementation/work-items.json`. Track implementation in the application workspace, with real child commits, a workspace snapshot, executed checks, and an evidence record. Completing a reading exercise is not enough to mark a product item complete. Never infer completion from a file's presence alone.

For a cross-repository change, identify producer contracts first, update consumers, test applications independently, commit child repositories, then update the workspace pins. Record blocked prerequisites explicitly. Preserve failed experiment output alongside the eventual fix.

Foundations

Reading route: Chapters 1, 2, 3, 4, 5.

Item	Implementation outcome
000	Repository audit, toolchain locks, and architectural decisions
001	Go workspace and modular process skeletons
002	Local development infrastructure without mandatory Kubernetes
003	Database migrations, query generation, and persistence conventions
004	Versioned HTTP API and generated client contract

Item	Implementation outcome
005	Angular application shell and design foundation
006	Shared integration fixtures and CI quality gates
007	Execution state machines and trust-boundary contracts

Identity

Reading route: Chapters 6, 17.

Item	Implementation outcome
008	Installation bootstrap and secure local authentication
009	Organizations, projects, memberships, and explicit RBAC
010	Append-oriented audit events and administrative history
011	Encrypted scoped secret storage and key management
012	Runner pools, enrollment tokens, and capability policies
013	Scoped API tokens and initial CLI authentication
014	Administrative settings and secret-management user experience
015	Identity and secret-security acceptance gate

Authoring

Reading route: Chapters 7, 8, 20.

Item	Implementation outcome
016	Versioned pipeline YAML schema and bounded parser

Item	Implementation outcome
017	DAG compiler and dependency validation
018	Typed parameters and restricted conditions
019	Immutable run snapshots and definition version history
020	Manual triggering, idempotency, and run creation API
021	Freestyle jobs using the same pipeline model
022	Pipeline editor, graph viewer, and validation feedback
023	Pipeline compiler and authoring acceptance gate

Scheduling

Reading route: Chapters 9, 10, 19.

Item	Implementation outcome
024	PostgreSQL queue and atomic job leasing
025	Authorized runner matching and scoped claim API
026	Heartbeats, fencing, and idempotent result acceptance
027	Failure classification, retry budgets, and reconciliation
028	Cancellation, deadlines, and timeout propagation
029	Concurrency limits, capacity accounting, and resource locks
030	Durable events, live state updates, and queue explanations
031	Scheduler crash and concurrency acceptance gate

Execution

Reading route: Chapters 11, 12, 13.

Item	Implementation outcome
032	Outbound runner service and durable local identity
033	Container executor with explicit isolation limits
034	Deterministic and credential-safe Git checkout
035	Sequential steps and bounded command lifecycle
036	Durable log segments, redaction, and upload resumption
037	Runner restart reconciliation and orphan cleanup
038	Run detail screen and resumable live log viewer
039	First real CI job and execution reliability gate

Source automation

Reading route: Chapters 8, 16.

Item	Implementation outcome
040	GitHub App connection and signed webhook ingestion
041	Branch and pull-request discovery with trust policies
042	GitHub checks, commit statuses, and delivery reconciliation
043	Scheduled runs, cron timezones, and catch-up policy
044	Bounded matrix jobs and conditional fan-out

Item	Implementation outcome
045	Automation timeline, run comparison, and actionable history
046	Scoped notifications and safe outbound delivery
047	Git automation and trigger-security acceptance gate

Build outputs

Reading route: Chapters 14, 15.

Item	Implementation outcome
048	Artifact upload, integrity, and storage abstraction
049	Artifact retention, secure downloads, and explicit job handoff
050	JUnit reports and test-result visualization
051	Dependency cache with trust-aware namespaces
052	BuildKit image builds and immutable registry outputs
053	Registry build cache and measurable build performance
054	SBOM and provenance records for release artifacts
055	Artifact, cache, and build acceptance gate

Delivery

Reading route: Chapters 17, 18, 19.

Item	Implementation outcome
056	Environments and server-enforced deployment policy

Item	Implementation outcome
057	Deployment records, orchestration, and uncertain outcomes
058	Approval gates with immutable subject binding
059	Restricted SSH deployment executor
060	Container release rollout, health checks, and controlled rollback
061	Opt-in workload OIDC and short-lived cloud credentials
062	Environment dashboards and deployment audit experience
063	Deployment safety and recovery acceptance gate

Operator experience

Reading route: Chapters 20, 21.

Item	Implementation outcome
064	Complete operational CLI with streaming and safe exits
065	Operational dashboards and efficient search
066	Operational quotas and abuse-resistant limits
067	Versioned reusable templates and isolated action contracts
068	Limited Jenkins migration analyzer and explicit unsupported cases
069	Guided onboarding and operator diagnostics
070	Accessibility, responsive behavior, and browser reliability
071	Feature-complete self-hosted acceptance and migration demo

Release hardening

Reading route: Chapters 22, 23, 24.

Item	Implementation outcome
072	OpenTelemetry, Prometheus, and actionable alerts
073	Backup, restore, and recovery consistency
074	Multi-scheduler hardening and safe schema upgrades
075	Self-hosted packaging, TLS topology, and runner installation
076	Release build pipeline and software supply-chain checks
077	Security regression, fuzzing, and release threat-model review
078	Load, chaos, and operational endurance verification
079	Self-hosted release-candidate audit and repository handoff

Evidence attached to a completed item

Record the exact source commits for changed repositories, the workspace commit that pins them, the contracts/schema version, commands executed, exit statuses, output paths, and the acceptance decision. Use real immutable references, not example hashes. A release build also needs independently resolvable dependencies and confirmed output digests.

A blocker is not a failed project. It is a precise statement that a necessary prerequisite is missing: a published dependency, a test database, an authorized target, a review decision, or a runtime capability. Resolve that prerequisite or revise the scope explicitly. Do not silently skip it while retaining the same completion claim.

Repository ownership rule

Use Chapter 3's repository boundaries when assigning ownership; the workspace records the tested composition.

Appendix C. Acceptance gates and failure experiments

Gate 1: source and build independence

From a clean environment, retrieve the workspace and all pinned child commits using the access level promised to readers. Build each application independently. For Go consumers, disable local workspace resolution. For the web, install from its own lockfile and verify its generated client receipt. Build each container from its declared context. Fail the gate if sibling source or unpublished dependencies are required without a documented verification path.

Evidence should identify dependency versions, commits, generator versions, and the clean-environment procedure. A successful build in the maintainer's long-lived checkout is insufficient.

Gate 2: identity and policy

Create two organizations, two projects per organization, and principals with deliberately different roles. Attempt cross-scope reads, mutations, log subscriptions, signed downloads, secret redemption, and runner-pool selection. Revoke access while a session and a pending run exist. Verify that client-side visibility is never the only denial mechanism.

Exercise runner enrollment twice with the same one-use credential. Rotate and revoke the resulting identity. Verify that browser, CLI, and runner credential classes cannot be substituted for one another. Check that audit events contain useful identities without plaintext secrets.

Gate 3: compilation and immutable intent

Submit missing dependencies, cycles, duplicate identifiers, oversized configurations, unknown keys, wrong parameter types, and excessive matrices. Verify that no partial runnable graph is committed. Compile the same accepted input repeatedly and compare its normalized identity. Change a branch or toolchain alias after compilation and confirm that the existing run keeps its frozen inputs.

Change a permission after compilation and verify that dispatch can become more restrictive. The snapshot must preserve intent without preserving revoked authority indefinitely.

Gate 4: queue concurrency and ownership

Run multiple real schedulers against PostgreSQL. Compete for one job, then many jobs with shared capacity limits. Inject deadlocks, transaction aborts, statement timeouts, and connection loss after commit. Verify current-claim uniqueness and atomic capacity reservation.

Expire a lease without stopping its worker. Ensure the job becomes uncertain rather than automatically safe to repeat. Deliver an old completion after an authorized retry and verify that it cannot replace the new attempt. Identical terminal reports should remain idempotent; conflicting reports should remain visible as conflicts.

Gate 5: executor and storage boundaries

Use the actual selected container runtime on a dedicated test host. Test resource limits, controlled mounts, network policy, image policy, timeout escalation, process exit observation, and cleanup ownership. Restart the runner during execution and reconcile its journal against runtime objects.

Interrupt log and artifact uploads after bytes arrive but before metadata commits. Retry identical segments, submit conflicts, exceed quotas, and use expired upload capabilities. Test unsafe archive paths and expansion limits. Confirm that unrelated projects and attempts cannot finalize or read the objects.

Gate 6: source and credential security

Send correctly and incorrectly signed webhooks, replay delivery IDs across two API instances, and reuse an idempotency key with different content. Verify that acknowledgements follow durable receipt creation. Test restricted contribution builds with no production credentials and no trusted-cache write path.

For workload federation, validate the actual provider configuration in a dedicated test environment. Test wrong audience, subject, issuer, expiry, and signing key. A local mock is not sufficient evidence for this gate. Record the provider-side policies and the resulting effective permissions without publishing credentials.

Gate 7: deployment uncertainty

Use a disposable target and a uniquely identified release. Change the artifact or configuration after approval and verify rejection. Compete for the same target from two runs. Disconnect before mutation, during mutation, and after the target acts but before acknowledgement. Verify that uncertain state keeps the target reservation.

Reconcile from a trustworthy target observation, then perform the documented next action. Test rollback compatibility and the point beyond which data changes cannot be reversed by changing application images. Preserve the complete operation history.

Gate 8: interface and operations

Test the browser with duplicated, missing, and out-of-order events; revoked access; large logs; malicious display text; and keyboard navigation. Verify CLI stdout/stderr behavior, exit statuses, context selection, and stale previews.

Restore a real backup with object data and required keys into an isolated environment. Keep dispatch paused, reconcile leases and targets, and document the controlled resume. Exercise the supported binary-version matrix during schema expansion and contraction. Measure load using a stated workload and record recovery after the load ends.

Release decision

The release record should name which gates ran, which evidence belongs to the exact source snapshot, what was skipped or blocked, and which limitations remain. Do not translate “not observed to fail” into “verified.” Public release notes must match the evidence, including the supported trust model and deployment scope.

These gates are requirements for a future full platform. The book's preparation evidence covers its own publication build and isolated Go laboratory, not a completed product deployment.

Appendix D. Glossary

Approval subject

The exact project, environment, artifact, configuration, and policy identity authorized by an approver.

Artifact

A finalized output with ownership, producer identity, digest, and retention policy.

Attempt

One concrete execution of a logical job; retries create new attempts.

Backpressure

A bounded response to a producer generating data faster than the consumer can accept it.

Build context

The explicitly supplied source material visible to an image build.

Cache

An optional optimization whose absence must not change correctness.

Capability

A narrowly scoped permission or token for one operation, resource, or attempt.

Claim

An atomic assignment of eligible work to a particular runner attempt.

Compiler

The component that validates pipeline input and produces a frozen execution plan.

Control plane

The services that decide, authorize, persist, and expose system behavior rather than execute repository code.

Cursor

A stable position used to resume reading ordered events or log segments.

DAG

A directed acyclic graph; its dependency edges do not contain a cycle.

Deployment

A tracked change of a target environment to an identified release.

Digest

A content-derived identifier; useful for integrity, but not proof of producer trust.

Execution envelope

The complete bounded runtime description handed to an executor.

Fence

A generation identifier checked to reject stale ownership at a boundary.

Gitlink

The superproject tree entry selecting a commit in a submodule.

Heartbeat

A current owner's request to renew its time-limited control lease.

Idempotency key

An identity under which identical request retries map to one accepted intent.

Inbox

Durable receipt and processing state for an incoming external event.

Job

A logical schedulable node whose attempts determine its outcome.

Lease

Time-limited control ownership; not proof that physical execution stops on expiry.

Outbox

A durable record of external delivery work committed with the state change that caused it.

Pool

An authorized grouping of runners with defined capabilities and access rules.

Provenance

Recorded information about how an output was produced and which inputs and builder participated.

Reconciliation

Establishing current reality after observations, ownership, or transport become uncertain.

Release manifest

A record connecting source commits, built outputs, versions, and verification evidence.

Run

An immutable execution request with source, inputs, compiled graph, and history.

Snapshot

A fixed representation of state or intent at a defined revision.

Step

An operation within a job, with defined workspace and exit semantics.

Superproject

A repository that composes independent child repositories using submodule references.

Target reservation

A concurrency control preventing incompatible operations on the same external target.

Trust domain

A scope within which code, credentials, caches, and execution authority are deliberately related.

Uncertain

A state in which the platform lacks enough evidence to assert the physical outcome safely.

Workspace overlay

A local development convenience that does not replace independent dependency declarations.

Appendix E. Primary sources

References were checked on **10 October 2026**. They support specific tool behavior, not a claim that the proposed platform is already implemented. The architecture, exercises, and acceptance gates are original project design. Exact supported dependency versions must be verified when deploying the product. External pages can change after this edition.

S01. Jenkins Pipeline

Existing pipeline concepts and their supported behavior.

[Official reference](#)

S02. Jenkins controller isolation

Separating build authority from the controller.

[Official reference](#)

S03. Git submodules overview

Gitlink identity and separate repository history.

[Official reference](#)

S04. Git submodule command

Registration, recursive checkout, and update behavior.

[Official reference](#)

S05. Go multi-module workspaces

Local development across multiple modules.

[Official reference](#)

S06. Go Modules Reference

Module identity, versions, and workspace resolution.

[Official reference](#)

S07. PostgreSQL SELECT

Row locking and queue-style SKIP LOCKED consumption.

[Official reference](#)

S08. Docker Engine security

Daemon authority and container isolation boundaries.

[Official reference](#)

S09. BuildKit upstream project

Builder capabilities and operational configuration.

[Official reference](#)

S10. Docker cache backends

Registry cache and backend-dependent support.

[Official reference](#)

S11. Docker build secrets

Temporary build-secret mounts.

[Official reference](#)

S12. Angular Signals

Reactive state primitives.

[Official reference](#)

S13. Angular security

Untrusted content, sanitization, and trust bypass risks.

[Official reference](#)

S14. HTML server-sent events

Event stream format and reconnection behavior.

[Official reference](#)

S15. GitHub webhook validation

Raw-byte HMAC signature verification.

[Official reference](#)

S16. GitHub webhook practices

Delivery identity, HTTPS, and acknowledgement practices.

[Official reference](#)

S17. GitHub secure-use reference

Untrusted source, privileged contexts, and self-hosted execution.

[Official reference](#)

S18. PostgreSQL transaction isolation

Snapshot semantics and transaction retries.

[Official reference](#)

S19. OpenID Connect for workloads

Federation and short-lived credentials.

[Official reference](#)

S20. PostgreSQL backup and restore

Backup strategies and recovery planning.

[Official reference](#)

S21. GitHub Pages custom workflows

Build artifacts, deployment permissions, and environment setup.

[Official reference](#)

S22. GitHub Pages publishing source

Selecting GitHub Actions as the publication source.

[Official reference](#)

S23. Creative Commons Attribution 4.0

Book-content reuse and attribution conditions.

[Official reference](#)

S24. MIT License

Permission notice for original code.

[Official reference](#)

License legal code

The complete legal code for the book-content license is available from [Creative Commons](#). The local licensing files state the scope of the grant and include the full MIT notice for original code.