THE ENGINEERING SERIES



INSIDE orch

Building and Operating a Docker Orchestrator in Go

Desired state. Runtime boundaries.
Failures you can explain.

Aleksandar Popovic

SOURCE-BASED ENGINEERING EDITION

Inside orch

Building and Operating a Docker Orchestrator in Go

Aleksandar Popovic

Copyright © 2026 Aleksandar Popovic. All rights reserved. No new public license is granted for the original manuscript. Source excerpts and third-party materials retain their existing rights and notices.

Source-based engineering edition · 1.0

Source revision `cbfc94ade5b57e449d6b73bf1485b1c5b1647022`. Observations describe the inspected and tested boundaries; no production certification is implied.

Contents

1. Preface 5
2. Chapter 1 - Why Build a Container Orchestrator? 9
3. Chapter 2 - A Repository Built for Reasoning 16
4. Chapter 3 - Boot the Lab and Deploy the First Service 24
5. Chapter 4 - Domain Models and State Machines 33
6. Chapter 5 - Service Specifications, Resources, and Ports 41
7. Chapter 6 - Follow One Request Through the Control Plane 49
8. Chapter 7 - HTTP Contracts and Compatible APIs 57
9. Chapter 8 - A CLI That Operators Can Automate 65
10. Chapter 9 - Persistence, Transactions, and Optimistic Concurrency 74
11. Chapter 10 - A Deterministic Scheduler 83
12. Chapter 11 - Reconciliation as Repeated Convergence 92
13. Chapter 12 - Idempotent Docker Operations and Drift 100
14. Chapter 13 - Agents, Heartbeats, and Task Reports 110
15. Chapter 14 - Health Checks That Respect Trust Boundaries 119
16. Chapter 15 - Events, Audits, and Notification Boundaries 128
17. Chapter 16 - Log Streaming Without Losing Control 137
18. Chapter 17 - Rolling Updates and Rollback Invariants 145
19. Chapter 18 - Safe Deletion and Resource Cleanup 154
20. Chapter 19 - Offline Nodes, Draining, and Rescheduling 163
21. Chapter 20 - Endpoint Discovery and Internal DNS 172
22. Chapter 21 - Ingress, Host Networking, and the Overlay Roadmap 181
23. Chapter 22 - Autoscaling Decisions and Missing Integration 189
24. Chapter 23 - Authentication, Authorization, and Node Trust 198
25. Chapter 24 - Encrypted Secrets and Configuration Redaction 207
26. Chapter 25 - Registry Credentials and Immutable Image Identity 217
27. Chapter 26 - Security Contexts, Admission, and Policy 227
28. Chapter 27 - Namespaces and the Limits of Multi-Tenancy 235

29. Chapter 28 - Quotas and Usage Accounting 243
30. Chapter 29 - Supply Chain Designs Without False Guarantees 251
31. Chapter 30 - GitOps, Drift, and Controlled Reconciliation 259
32. Chapter 31 - Finite Jobs and Time-Based Work 267
33. Chapter 32 - Local Volumes, Claims, and Workload Topology 276
34. Chapter 33 - Extension Interfaces and a Plugin Roadmap 285
35. Chapter 34 - Leader Election and the Durable HA Boundary 293
36. Chapter 35 - Upgrades, Migrations, Maintenance, and Retention 301
37. Chapter 36 - Fakes, Property Tests, Fuzzing, and Contracts 308
38. Chapter 37 - Failure Injection and Recovery Evidence 316
39. Chapter 38 - Metrics, Dashboards, and Actionable Signals 324
40. Chapter 39 - Performance and Scale Tests That Say What They Measure 332
41. Chapter 40 - Release a System You Can Honestly Operate 339
42. Appendix A - CLI, API, and Configuration Reference 347
43. Appendix B - Lifecycle States and Invariant Reference 356
44. Appendix C - Commit Atlas and Original Prompt Crosswalk 364
45. Appendix D - Worked Exercises and Design Solutions 374
46. Appendix E - Operator Runbooks and Measurement Contracts 382
47. Appendix F - Glossary, Bibliography, and Boundary Index 388

Preface

An orchestration system is most revealing when something interrupts its plans. A container starts but its status report disappears. A worker stops answering while its application remains reachable. An operator restarts a server and discovers that the running containers remember more than the new control plane. These moments expose the difference between a useful algorithm, an implemented package, and a working recovery protocol. This book studies that difference through one inspectable Go and Docker system: orch.

I wrote this edition for readers who want to understand the machinery behind an orchestration interface. Developers need to follow types and interfaces until a request becomes a runtime effect. DevOps engineers need to distinguish an accepted deployment from available application replicas and know what to inspect after a failure. Architects need to identify where authority lives, which invariants are local, and which guarantees require coordination across processes. Those interests overlap. An operator's unexplained failure often points to a developer's missing boundary, while an architectural promise becomes credible only when someone can test its recovery conditions.

One source edition, with an honest boundary

The source edition is orch commit `cbfc94ade5b57e449d6b73bf1485b1c5b1647022`. References use that full revision so readers can inspect the same files even as the application changes. The accompanying history contains ninety-one development prompts and 105 reachable commits. Prompts describe requested work; implementation and tests show what was built. Several early prompts were added retrospectively, and later work landed in batches. A completed development tracker is useful navigation, but it is not a transcript establishing that a laboratory ran or that a distributed guarantee holds.

The most important fact appears before the sophisticated features: the default server constructs an in-memory control plane. Its service, node, task and deployment records belong to the running process. PostgreSQL storage code, migrations and backup scripts exist, but their presence does not make the default API state durable. Leadership, autoscaling and reconciliation packages also need to be read with their callers in view. A package can provide a sound building block without being the implementation selected by the executable.

This boundary is an opportunity to learn rather than a detail to hide. We can examine deterministic planning without claiming every service uses that planner. We can study optimistic database assignment without pretending it reserves every resource globally. We can inspect Docker limit support while tracing whether the agent supplies the required fields. Every chapter asks both what a component does and how its behavior reaches the running system.

Three ways through the book

Everyone should read the opening chapters on scope, the laboratory, models and the request path. They establish the vocabulary of desired state, reported state, identity and process ownership. Later chapters repeatedly use those definitions instead of quietly changing what "running," "available" or "recovered" means.

The developer path follows API contracts, the CLI, persistence interfaces, scheduling, reconciliation, Docker operations and agent reports. Read it with the source open. Pay particular attention to the smallest function that decides a state transition and the caller that gives it authority. Exercises ask you to reason about stale snapshots, idempotency, concurrency and test boundaries before proposing changes.

The DevOps path follows deployment flow, probes, events, logs, rollouts, node loss, secrets, upgrades and observability. Its recurring question is what an operator can actually observe. Keep a worksheet with service, task, node, deployment and container identifiers. A successful

application request, a healthy agent heartbeat and a control-plane record are separate pieces of evidence; none should silently stand in for the others.

The architect path follows composition, state ownership, admission, tenancy, jobs, local storage and leadership. It emphasizes failure windows and tradeoffs rather than product checklists. Use the worked scenarios to decide where a stronger guarantee would need durable identity, atomic mutation, fencing or a different topology. Design-only topics remain useful when the missing implementation is stated precisely.

A laboratory that respects its neighbors

The running example is Atlas, a small application used to connect chapters. Its services, jobs and operational decisions become more interesting as the book progresses. The laboratory is disposable. Go commands run against a separate copy of the pinned application source, not against the website repository or the immutable research snapshot. Containers, temporary repositories, ports and databases belong to that laboratory and must have an explicit cleanup owner.

Start with deterministic package tests and local fixtures. They provide focused evidence without demanding an entire cluster. Docker and PostgreSQL experiments are additional boundaries, not decorative prerequisites. Some upstream tests reset database schemas, so only a database created for the experiment is acceptable. A shared development daemon also deserves care: never replace scoped cleanup with a command that deletes every container or volume on the host.

Each experiment states prerequisites, commands, observations, abort conditions and cleanup. An expected result describes what source inspection predicts; an observed result requires a recorded run. Missing tools or unavailable external services remain visible gaps. Reading can continue while a laboratory is blocked, but a release cannot claim that the blocked integration was verified.

How to read the evidence

Short source listings are unchanged excerpts with pinned file and line references. Adapted examples are labeled teaching extensions and live separately from the application baseline. Diagrams explain ownership and state changes; they are not permission to invent a connection absent from startup code. Tests inspected in source establish intended assertions. Only command records with scope, timestamps, exit status and logs establish execution.

Performance results require measurement context. Security findings distinguish source-derived concerns from demonstrated behavior. A fake runtime is valuable for deterministic faults, but it does not prove Docker, network or database behavior. Likewise, a controller reconstructed over the same memory object has not demonstrated a new server process recovering its state.

Exercises have worked answers, yet their purpose is to sharpen judgment rather than reward recall. Ask where identity survives, who can repeat a side effect, what happens between a write and its report, and which observation would falsify your explanation. By the final chapter, you should be able to describe both the system's useful mechanisms and its unresolved boundaries with equal precision. That is the foundation for evolving an orchestrator responsibly—and for operating the version that actually exists.

Chapter 1 – Why Build a Container Orchestrator?

A Small Cluster Is Still a Distributed System

At nine in the morning, the Atlas operator has three Docker hosts, an HTTP service, and a deployment checklist. The checklist says to pull the new image, stop the old container, start its replacement, and check an endpoint. At nine fifteen, a laptop connection disappears between the stop and start commands. Another operator repeats the checklist on a different host. The team now needs an answer to a question the shell history cannot settle: which version should be running, where, and who is responsible for getting it there?

That is a sufficient problem to study orchestration. It does not require thousands of containers. Multiple machines, delayed observations, and interrupted actions already create a distributed system. Atlas needs repeatable intent, identifiable work, and a way to compare desired state with reports from workers. The smallest useful controller must decide what happens after an action succeeds but its acknowledgement is lost. Adding a pleasant deployment command without answering that question merely moves the uncertainty behind a different interface.

By the end of this chapter, you should be able to trace a service request through the actual server and agent, identify the owner of each state, and decide which engineering claims the default binary supports. You should also be able to explain why building a small orchestrator can be a useful learning or internal-tool exercise without presenting its existence as a production recommendation. The rest of the book follows those questions through source rather than treating a list of modules as a finished platform.

Atlas uses the name `atlas-api` for the continuing case study. Its first job is a stateless HTTP endpoint on disposable infrastructure. Later chapters introduce update failures, disconnected workers, resource constraints,

and finite jobs. This identity is a teaching scenario, not an assertion that this repository has operated a particular commercial cluster. We can reason concretely about ownership and failure without inventing deployment history, adoption numbers, or availability measurements.

An orchestrator is useful when repeating a desired result is more valuable than remembering a sequence of imperative commands. “Keep two replicas of this service at this version” survives an operator leaving the terminal. “Run these six commands” does not specify what to do halfway through a failure. The distinction also changes how errors should be reported: an accepted service is a stored intention, while a running task is a later observation. The API can succeed before an image pull fails.

For a small team, this separation can clarify operational responsibilities. It can also create responsibilities the team previously delegated to an established platform: identity, authorization, capacity, durable state, safe replacement, observability, upgrades, and recovery. Evaluate those obligations before choosing to run a custom system. A source-based book can teach how they fit together; it cannot transfer a production guarantee from the author's confidence to your environment.

Who Owns Desired and Observed State?

The domain begins with five nouns. A service describes desired workload configuration. A task identifies one execution attempt for a service version. A node describes a worker and its reported capacity and status. An agent executes work assigned to that node. A Docker container is the local runtime object the agent creates or adopts. These nouns are related, but none is a synonym for another. Deleting a service record is not equivalent to stopping a container on a disconnected worker.

The ServiceSpec definition contains image, replicas, environment, resource requirements, security context, and other declared inputs. The server stores a service with an ID and deployment version. A task then binds an execution identity to a node and tracks desired and actual status. The

agent polls its task assignment and performs the image/container operations through an interface. The container creation call is the concrete bridge from task identity to a local Docker request.

Object	Owner of the relevant decision	Evidence it does not supply by itself
Service	Operator intent accepted by control plane	A container is running
Task	Server assignment and agent reports	Remote side effect happened exactly once
Node	Registration, heartbeat and monitor	A silent worker has stopped serving
Agent	Local execution and observations	Cluster state survives server replacement
Container	Docker daemon on one host	Other replicas agree with this version

Consider one Atlas replica. The CLI submits a service. The API checks the request and invokes the control-plane service. The server creates or reconciles task state. The worker polls the assignment, pulls the image, creates a labeled container, starts it, and reports progress. A health observation may follow. Each step has an identifiable owner and a failure boundary. The word “deployment” spans the sequence; it does not turn the sequence into one transaction.

The agent does not decide that the service needs three replicas instead of two. The scheduler does not start a container. The Docker adapter does not know whether a deployment is commercially successful. Keeping those boundaries narrow makes errors easier to investigate. When Atlas has a task but no container, inspect the agent/runtime path. When there is no task, investigate control-plane intent and placement. When Docker shows a container but the API does not, investigate observation, identity, and state ownership before creating a replacement.

There is also a difference between desired state and reported state. A report can be stale, delayed, or lost. The server's status is a record of what it accepted, not an independent measurement of the worker at the

instant you read it. Later laboratories compare API state, agent reports, Docker inspection, and a direct endpoint request. Agreement across those observations is stronger evidence than a single green status field, although it remains scoped to the observation window.

What the Default Binary Actually Runs

Start the architecture review at the composition root. In server startup, the selected state owner is explicit:

```
        controlPlane :=
controlplane.NewMemoryService(controlplane.WithSecretEnvelope(envelo
pe), controlplane.WithClusterPolicy(cfg.ClusterPolicy),
controlplane.WithRetentionConfig(cfg.Retention))

controlPlane.SetNotificationDispatcher(notifications.NewDispatcher(c
ontrolPlane, nil))
        serverMetrics := metrics.NewServer()
```

Exact source: cmd/orch-server/main.go:146–148.

This is **runtime-integrated** behavior: ordinary startup constructs `controlplane.NewMemoryService`, attaches notification dispatch, and creates server metrics. The memory service allocates maps for nodes, services, tasks, deployments, secrets, and additional objects. Those maps belong to the process. A database URL does not change the selected constructor. The optional startup schema check opens PostgreSQL for compatibility inspection and closes the runner; it is not a storage injection mechanism.

The server does start useful loops. The GitOps, cron and scheduler construction is followed by usage snapshots, retention pruning, the rollout controller, and node monitoring. Optional DNS and a separate debug listener have their own configuration gates. This is more specific than saying “the server runs controllers.” It tells us which controllers, which store they share, and which effects disappear if their goroutine stops.

Several substantial packages are absent from this graph. PostgreSQL store code is an **implemented package**, not the active server service. Standalone reconciliation is another implemented package; default memory mutations perform a different immediate reconciliation path. Leadership and autoscaling packages are not constructed by ordinary startup. Calling them implemented is fair. Saying two default server processes form an HA control plane, or an autoscaling policy automatically enables a metrics-driven loop, would add behavior the composition does not provide.

Even a constructed scheduler needs a caller-sensitive explanation. Immediate service reconciliation chooses ready, volume-eligible nodes and creates assigned tasks. The independent scheduler plans pending tasks. Ordinary service tasks can therefore be assigned before the planner encounters them. An algorithm's presence does not mean every assignment passed through it. We will trace this difference before teaching resource or placement guarantees.

The history helps explain how these boundaries arose. Architecture conventions established the project shape. The first REST API selected an in-memory implementation. Original development prompts 00 and 26 describe architectural intent and a review request, but early prompt files were committed after the June implementation. They are retrospective associations, not proof that a particular instruction was executed before a particular commit.

The First Capability Boundary

The first Atlas failure window is a server process replacement. Before the interruption, maps contain the service and its tasks. During the interruption, Docker may continue running containers on workers. After the replacement, a new memory service has fresh maps. The continuing existence of a local container does not recreate the missing service intention. Conversely, an empty API response does not prove no old container remains. Those are different observations about different owners.

This explains why a readiness endpoint cannot establish recovery. The health and readiness handlers return fixed success responses. They show that the selected handler is reachable. They do not query durable cluster state, compare replicas, establish leadership, or verify that old credentials remain valid. Atlas should record reachability and workload recovery separately rather than using the former as a substitute for the latter.

There is a precise engineering choice here. For disposable learning, a memory-backed control plane makes experiments cheap and state ownership visible. For a system whose desired state must survive process loss, that composition is insufficient. Replacing it requires more than choosing a database library: service semantics, all required object types, transaction boundaries, credential recovery, migrations, and process restart tests must agree. The book studies those requirements without silently patching the baseline to claim they already exist.

Use the capability question at every new subsystem: where is this behavior connected to the running system? Then ask who owns its state, what side effect it performs, and what evidence would distinguish success from a convincing status report. A package, schema table, test name, or completed progress tracker can help you navigate. None independently answers all four questions. This discipline is valuable even if you eventually choose a different orchestrator.

Atlas Lab: draw the actual boundary

Prerequisites are the immutable source snapshot, Python, and a disposable directory for any later builds. No daemon or database is needed for this first source experiment. From the website repository root, read the lock and inspect the composition without running application commands inside the website source tree:

```
python3 -c 'import json; print(json.load(open("books/inside-orch/
evidence/source-lock.json"))["baseline_sha"])'
sed -n '142,234p' books/inside-orch/evidence/orch/cmd/orch-server/
main.go
```

Draw the server, memory owner, API, active loops, worker, and Docker daemon. Put PostgreSQL beside the graph as a package/migration boundary, without an arrow that pretends it owns normal server state. Annotate the worker connection as polling/reporting, and annotate container existence as local daemon state. Success means each arrow has a constructor or call site; reject a diagram whose only support is a roadmap sentence.

The source map and exact SHA were mechanically checked during this edition's research. The process-restart and real-container comparisons are separate experiments, not results of these two commands. Record your own UTC time, command exit status, source SHA, and diagram notes. This source-only step creates no containers or services; cleanup removes only your drawing or notes if desired. Retain the map as the initial Atlas evidence artifact.

Exercises

1. Explain why an accepted service and a healthy replica are separate facts. Name the owner of each fact.
2. Find the constructor that owns live state. Explain why a successful PostgreSQL migration does not change its selection.
3. An operator sees `/readyz` succeed after restarting the server and declares recovery complete. Identify two observations still missing.
4. Decide whether Atlas should use this baseline for a disposable training cluster or a payroll workload requiring restart-safe desired state. Justify the decision using composition rather than product labels.

Worked answers connect each decision to its specific source boundary. The architecture decision for the next chapter is to follow interfaces and constructors before searching for additional features. Claims are indexed in the chapter evidence ledger; CAP-001, CAP-003, CAP-004 and DOC-001/002 remain controlling boundaries for later explanations.

Chapter 2 – A Repository Built for Reasoning

You will learn to find active interfaces from their callers, distinguish deterministic package fixtures from infrastructure tests, and interpret cancellation as a boundary rather than a rollback guarantee.

The Actual Tree Versus the Intended Layout

The Atlas developer wants to reproduce an assignment problem before changing production settings. A directory named `scheduler` looks promising, but searching for “runtime” finds a different tree from the architecture sketch. Following the sketch literally would send the developer to a directory that does not exist. The first useful repository skill is therefore not memorizing a preferred layout. It is locating the path a real caller takes and distinguishing that path from the structure somebody intended to build.

This chapter teaches you to navigate the actual packages, find side-effect interfaces, and explain how cancellation and deterministic fixtures constrain a design. You will run a small test with an observable assertion rather than start the whole system to answer every question. The Atlas artifact is a source index that identifies a constructor, dependency, state owner, and effect for each subsystem. Chapter 1's boundary map becomes a map you can use while debugging.

The application's intended layout places future runtime adapters under an `internal/runtime` hierarchy. Current code instead uses `internal/docker`, which contains the Runtime interface, EngineRuntime, FakeRuntime, and instrumentation wrapper together. Health probing lives in `internal/health`; orchestration of repeated probes and threshold counters lives in the agent. Do not rename these paths in an explanation merely to make the tree match an architectural vocabulary. The existing names reveal where responsibilities actually accumulated.

The executables are similarly concrete. `cmd/orch` delegates to Cobra composition. `cmd/orch-server` selects the memory service and starts its controllers/listeners. `cmd/orch-agent` assembles a Docker adapter, runner, and log listener. A fourth executable, `cmd/orch-loadtest`, constructs its own memory service for synthetic workloads. The presence of the load tool is useful, but its observations are not observations of a network-connected production server. Navigation begins with these entrypoints because they decide which packages become active behavior.

A small package can also be less significant than its name suggests. `internal/task/task.go` wraps a Task in DesiredState, and `internal/node/node.go` contains an older local Record/Status wrapper. The active lifecycle contracts live in `pkg/types`, while node monitoring lives in another file in `internal/node`. Import use and constructor calls matter more than the apparent importance of a filename. A reader should resist building a conceptual model around a wrapper that no relevant path consumes.

The practical index is intentionally short. For scheduling, record New, RunOnce, Plan, and persistAssignment. For agent execution, record NewRunner, Run, ensureTask, and the Runtime interface. For persistence, record NewPostgresStore, store.WithTx, and the SQL assignment guard. For each, add a caller reference. This is more useful than reproducing every directory because it gives the next investigation a starting point and a question about state or effects.

Interfaces at the Side-Effect Boundary

The Docker Runtime interface makes the agent's external effects explicit. It pulls images, creates/starts/stops/removes containers, inspects/list-labeled containers, and streams logs. Its operations accept context and

typed inputs rather than arbitrary shell commands. The interface tells us which effects an agent fixture must simulate. It does not say every simulation is equivalent to a real Docker daemon.

```
type Runtime interface {
        PullImage(ctx context.Context, image string, auth
*RegistryAuth) error
        CreateContainer(ctx context.Context, spec ContainerSpec)
(ContainerID, error)
        StartContainer(ctx context.Context, id ContainerID) error
        StopContainer(ctx context.Context, id ContainerID, timeout
time.Duration) error
        RemoveContainer(ctx context.Context, id ContainerID, force
bool) error
        InspectContainer(ctx context.Context, id ContainerID)
(ContainerStatus, error)
        ListManagedContainers(ctx context.Context, labels
map[string]string) ([]ContainerStatus, error)
        StreamLogs(ctx context.Context, id ContainerID, opts
LogOptions) (<-chan LogLine, <-chan error)
}
```

Exact source: internal/docker/docker.go:47–56.

Now follow that dependency through composition. `cmd/orch-agent.run` constructs EngineRuntime from the environment, wraps it with agent metrics, and injects it using WithRuntime. NewRunner initially creates bookkeeping and a health checker, while Run fills in missing client/runtime dependencies if they were not supplied. A test can supply FakeRuntime without changing the agent's lifecycle logic. This is a usable seam, not just an interface declared for aesthetic reasons.

The idempotent fake-runtime fixture calls CreateContainer twice with the same task, checks that both calls return one identity, starts it, inspects its labels/state, and verifies the recorded operation sequence. The useful assertion is not “mock called.” It is that repeated local creation for the same task did not produce two container identities in this model. A real-daemon test must check the corresponding adapter behavior separately. The fake has a mutex and a concurrent-create test, but neither proves a distributed exactly-once protocol.

Consumer-owned interfaces also keep dependencies small. Scheduler.Store expresses the reads and assignments the scheduler needs. It does not require the entire API service, a Docker client, or a global repository object. Plan receives a value snapshot and returns proposed assignments; RunOnce performs I/O around it. This separation lets Atlas test insufficient capacity or label mismatch without a daemon. It also makes an important limit visible: a deterministic plan can still be stale when another actor writes before persistence.

A dependency can be replaceable without being connected in the binary you are using. The standalone reconciler has a Store and LeaderLock seam, but ordinary server startup never constructs that reconciler. Leadership has another elector interface and wrapper; its existence does not activate HA. When reading a new interface, find both its implementation and its injection point. If the latter is absent, classify it as an implemented package rather than silently invent a runtime caller.

Cancellation and Time as Dependencies

Atlas's first shutdown question is simple: what stops when the owner cancels the context? The main programs derive a signal-aware context and pass it downward. Server background loops select on cancellation and stop their tickers. The HTTP listener receives a separate bounded shutdown context. A Docker or HTTP operation can receive the caller's cancellation through its interface. These are different mechanisms serving one ownership rule: work should not outlive its declared owner accidentally.

Cancellation is cooperative, not a rollback transaction. Suppose an image pull succeeds and the context is canceled before the status report. The daemon may retain the image, and a container may already exist after a later interruption. Stopping the goroutine does not erase that side effect. Recovery must rediscover identity and reconcile it. The interface helps propagate cancellation, while labels/idempotency help resolve work that crossed the side-effect boundary before cancellation arrived.

Time deserves the same care. A controller with a clock can test a timeout without sleeping until a real wall clock happens to cross a boundary. The reconciler constructor stores a now function and interval; cron explicitly accepts a Clock. The agent has heartbeat/probe jitter and timer logic. These choices reduce synchronized behavior in a live system but can complicate tests unless randomness and elapsed time are controlled at the seam relevant to the assertion.

An invariant test should assert the decision, not the scheduler luck of the test process. For example, if two nodes tie, Plan sorts and breaks ties deterministically. A fixture can reverse input order and expect the same node identity. If a timeout changes desired state, an injected clock should advance to the boundary explicitly. “Sleep briefly and hope the controller has run” leaves the test sensitive to host load and says little about why the state changed.

There is also a meaningful distinction between stopping one controller and restarting a process. Recreating a controller against the same memory service preserves all domain maps. A fresh server process does not. Keeping state ownership in the index prevents an appealing restart test name from crossing that boundary. Test scope is part of the design explanation, not a footnote attached after a green result.

The initial scaffold created command entrypoints and package boundaries, while repository conventions documented intended responsibilities. Original development prompts 00, 01 and 21 are useful statements of intent, but their retrospective associations do not prove a transcript or a successful run. Later code placed Docker implementations and fakes together rather than adopting every intended path. Preserve the historical filename when discussing the scaffold and the current filename when explaining a caller. That small editorial distinction prevents a reader from trying to execute a command against a directory that existed only in a plan.

The First Focused Test

The pinned module file requires Go 1.25.0. Use the toolchain matching the baseline, record its executable and GOROOT, and execute application commands only in a disposable source copy. If source is exported without Git history, disable VCS stamping explicitly rather than treating a missing `.git` directory as a failed application feature. Record that build flag so a later reader can reproduce the environment difference.

The Makefile is convenient but has a subtle scope change: make lint runs golangci-lint when installed and otherwise falls back to gofmt and vet. A successful fallback is not an executed golangci-lint run. Likewise, normal go test can skip external database/Docker suites. Commands with identical friendly names can exercise different tools depending on the environment; capture the actual command and prerequisites rather than report only "quality checks passed."

Atlas first inspects PlanAssignments and the idempotent fake runtime test. These fixtures have finite inputs and concrete expected identities. From the disposable Orch source root, with the pinned toolchain selected, run:

```
go version
go build ./...
go test -count=1 -v ./internal/scheduler ./internal/docker \
  -run 'TestPlanAssignments|
TestFakeRuntimeRecordsOperationsAndCreatesIdempotently'
```

In this edition the isolated build and focused tests were actually executed successfully. The focused record is `validation/labs/ch02/focused-tests.json`; it stores the exact toolchain path, UTC timestamps, environment keys, exit status, and output. It establishes the assertions of these selected package fixtures. It does not establish real Docker pulls, PostgreSQL transactions, all possible randomized cluster states, or end-to-end resource enforcement.

Read a passing fixture in terms of its counterexample. “Insufficient resources” should yield no assignment for that input. “Deterministic tie breaking” should choose the stable node. Repeated CreateContainer should return the same fake identity. If a selected test name matches nothing, Go can still exit successfully, so inspect verbose output for the expected test names. A useful record includes assertions that ran, not merely a process exit code.

The failure window for this chapter is an injected create success followed by a lost report. The seam allows a test to preserve the fake container while retrying the agent's operation. That is a narrow and useful experiment. It does not reproduce Docker daemon crashes, network partitions, or server state loss. Those require different fixtures or real processes because they alter different owners. Choosing the smallest faithful seam is more rigorous than starting every dependency and hoping the resulting complexity improves evidence.

Cleanup is correspondingly small. Focused tests own their in-memory fixtures and temporary directories. Remove only disposable build outputs you created; never format or compile into the immutable snapshot. Keep the source index and execution record. If local listener or toolchain prerequisites are unavailable, mark that experiment blocked and preserve the source walkthrough. Do not substitute a test title for its missing run.

Exercises

1. Locate Runtime, EngineRuntime, FakeRuntime, and the executable injection point. Explain why the four references are stronger than a directory name.
2. Explain what PlanAssignments can establish without Docker and which ordinary service path still needs separate examination.
3. A controller is reconstructed against the same MemoryService and the test passes. Which restart claim is supported, and which is unsafe?
4. make lint succeeds on a host without golangci-lint. Draft an honest validation statement and name the remaining check.

Worked answers justify each scope decision. Atlas now has a source index and a reproducible package check. The next experiment introduces processes and containers only for questions those smaller seams cannot answer.

Chapter 3 – Boot the Lab and Deploy the First Service

Pinned Prerequisites and Installation

Atlas has a source map and passing package fixtures. The next question needs actual processes: can an operator submit a service, distinguish accepted intent from task execution, and remove the resulting workload without touching somebody else's containers? This chapter follows the upstream local-development path and makes its assumptions visible before starting it. You will learn to build the pinned binaries, inspect node/service/task state independently, and keep cleanup tied to the resources the experiment owns.

Use a disposable checkout or copy of source revision `cbfc94ade5b57e449d6b73bf1485b1c5b1647022`. The immutable evidence directory is for reading, not running scripts or writing build outputs. Go 1.25.0 is the declared baseline. Docker Engine access and Docker Compose are required for the upstream local stack; curl is used by the startup script. PostgreSQL runs in Compose, but its presence does not make the default control plane durable. The three processes have different state owners even when their containers share one Compose project.

The installation guide documents platform archives for Linux, macOS and Windows on amd64/arm64. This edition's source lock establishes the code revision, not availability of every advertised release archive. Building from the pinned source avoids assuming a moving installer selects the same revision. Record the compiler version, command and source hash. An exact source build is a different evidence statement from “the installer downloaded the latest release.”

The source-only build command is straightforward, provided it runs in the disposable Orch root:

```
go version
go build -o ./bin/ ./cmd/orch ./cmd/orch-server ./cmd/orch-agent
./bin/orch version
```

For an exported snapshot without Git metadata, select the documented build environment with VCS stamping disabled. Do not edit go.mod to fit whatever compiler happens to be installed. The edition's build records preserve the exact downloaded toolchain and environment. Their successful compilation establishes that baseline on this host, not all supported platforms or real daemon operation.

Before starting Compose, reserve host ports 8080, 8081 and 5432 or stop and choose the separately scoped direct-process alternative. Do not kill unrelated listeners to force the example to run. Inspect the Compose file and startup script first. A running local website on another port is irrelevant; an existing database on 5432 is a concrete collision. The startup script has an unbounded readiness loop, so a failed server can otherwise leave an operator waiting indefinitely.

One Local Server and One Worker

The startup script changes to its own source root, copies .env.example when .env is absent, and starts postgres, orch-server and orch-agent with the local-orch profile. That script does not run migrations. It waits for the server's fixed ready endpoint, then prints listener addresses. Those messages describe intended endpoints; they are not proof the worker registered or the application is healthy.

The current main Compose fixture is `compose.yaml`; historical development commits used docker-compose.yml before a later rename. Use an explicit unique COMPOSE_PROJECT_NAME so project-level shutdown identifies your stack. The script's .env creation belongs in the

disposable copy and should contain synthetic development settings only. Do not copy a real environment file into the manuscript or execution evidence.

```
export COMPOSE_PROJECT_NAME=inside-orch-atlas-local
./scripts/dev-up.sh
./scripts/migrate-up.sh
./bin/orch --server http://127.0.0.1:8080 node ls
```

The startup decision itself is small enough to inspect exactly:

```
if [[ ! -f .env ]]; then
  cp .env.example .env
  echo "Created .env from .env.example"
fi

docker compose --profile local-orch up --build -d postgres orch-
server orch-agent

echo "Waiting for orch-server..."
```

Exact source: scripts/dev-up.sh:7–14.

A bounded wrapper is useful when executing this procedure unattended. The wrapper should enforce a deadline and preserve server/agent logs if readiness never arrives, rather than report the last printed message as success. It must also retain the unique project name so a timeout cleans up only the attempted stack. Adding a deadline is a book-side operational adaptation; it does not mean the upstream script already implements one.

The migration command is separate because database schema setup and active state ownership are separate operations. Read its URL/default behavior before invocation and ensure it targets the owned Compose database. The server environment contains DATABASE_URL, but ordinary startup still selects NewMemoryService. A successful migration demonstrates SQL setup against that database. It does not change which service implementation handles POST requests.

The worker container mounts the host Docker socket. Consequently, workload containers are created through that host daemon; they are not nested containers owned automatically by the Compose service lifecycle. The worker advertises its log-server address inside the Compose network so the server can proxy logs. This distinction matters during cleanup: removing the agent container is not itself a supported instruction to delete all previously created workload containers.

The fixture deliberately disables user JWT authentication and supplies development agent-registration/encryption values. These settings make a loopback learning environment convenient, but they are not production credentials or a deployment recommendation. Agent endpoints and user endpoints have different authentication paths. Keeping user auth empty does not mean an arbitrary token is valid for a node credential. Later identity chapters examine those paths; here, preserve the fixture's explicit trust assumptions.

Atlas first checks node state rather than deploy blindly. A successful node list request can return no nodes even while the API is ready. Inspect hostname, advertised address and ready status, then keep the returned node ID as part of the experiment record. If the agent never registers, investigate its server URL, Docker connection and bootstrap setting before treating a service's lack of containers as a scheduler problem.

The First Deployment Timeline

The upstream http-api fixture uses nginx:1.27-alpine, one replica, container port 80 and public port zero. Its label request is role=worker and its resource values are 100m and 128Mi. Public zero asks for a dynamic host port rather than binding every replica to one fixed host port. The upstream name stays `http-api` in this exact walkthrough; later Atlas manifests explicitly use `atlas-api` instead of silently pretending the source fixture had another name.

Validate locally before submitting it:

```
./bin/orch validate deployments/examples/http-api.yaml
./bin/orch --server http://127.0.0.1:8080 deploy deployments/
examples/http-api.yaml
./bin/orch --server http://127.0.0.1:8080 --output json service
inspect http-api
./bin/orch --server http://127.0.0.1:8080 --output json service ps
http-api
./bin/orch --server http://127.0.0.1:8080 events --service http-api
```

Local validation proves that this parser accepts the file. Server admission still checks current policy, references, namespace and quota. A successful deploy response proves accepted service state; it does not prove nginx has started. Read desired replicas from the service spec and actual status from matching tasks. Keep task IDs, node IDs and container IDs distinct. A transition from assigned through pulling/created/running describes reported progress, while an independent Docker inspection confirms the local object.

Set explicit rejection criteria before the first deploy: reject a missing registered worker, a service whose accepted spec differs from the intended manifest, a task assigned to an unexpected node, or a container whose managed labels do not match the recorded task. Also reject cleanup evidence collected only after a broad daemon-wide deletion. A convincing empty list after destroying unrelated workloads is a failed ownership experiment. These criteria make the demo assessable even when the expected happy path is interrupted.

Save the pre-scale snapshot before issuing scale. Otherwise a later list containing two tasks cannot tell you whether the original task survived or both were replaced. Likewise, preserve the pre-delete IDs so an empty service response can be compared with the exact local objects that previously existed. The evidence is a relationship across snapshots, not the attractiveness of one final table.

The normal memory path may create an assigned task immediately. Do not insert an imagined pending state just because the architecture document describes a standalone reconciler followed by the scheduler.

The agent then polls and acts. Depending on observation timing, a single service-ps request may miss intermediate states. Missing a short-lived state in a poll is not evidence the code skipped it. Use timestamped snapshots and accepted reports when reconstructing a timeline.

Scale only after establishing the first replica's identity:

```
./bin/orch --server http://127.0.0.1:8080 scale http-api --replicas
2
./bin/orch --server http://127.0.0.1:8080 --output json service ps
http-api
./bin/orch --server http://127.0.0.1:8080 logs http-api --tail 20
./bin/orch --server http://127.0.0.1:8080 scale http-api --replicas
1
```

The dynamic published ports let two replicas use distinct host ports on one worker. That does not prove all placement/resource policies were enforced. The agent path omits resource limits even though the manifest declares them; Chapter 5 measures that boundary. Logs can legitimately be empty before traffic reaches nginx. An empty log stream is not a deployment failure by itself, and a successful log proxy is not a direct health request to every replica.

The motivating interruption occurs after Docker creates a container but before the server accepts its created report. The task record can lag behind the daemon. An agent retry should rediscover the same task identity, not create a new unrelated replica because the human saw stale status. The package fake/adoption tests address part of this scenario; a real runtime observation must correlate labels and IDs. Do not manually run docker rm merely to make a stale API response look consistent.

The local environment commit introduced scripts and fixtures; the Cobra CLI commit introduced operator commands. The final pinned documentation update describes installation. These are engineering/history evidence, not stored proof that this complete demo ran in those commits. Original prompts 05,21,27,28 remain retrospective intent associations where appropriate.

Cleanup and the Limits of the Demo

Remove the workload through the server while the worker is still available:

```
./bin/orch --server http://127.0.0.1:8080 delete http-api
./bin/orch --server http://127.0.0.1:8080 --output json service ps
http-api
./scripts/dev-down.sh
```

Deletion marks desired cleanup rather than instantly proving every container disappeared. Wait for the owned task/container identities to reach removal, and independently inspect those IDs before shutting down the worker. The shutdown script performs Compose down for selected profiles. It does not enumerate and remove every container created through the host socket, and it does not use down --volumes. PostgreSQL volume retention and workload cleanup therefore need separate observations.

If the server fails before deletion, stop automatic experimentation and inspect only objects carrying your recorded task/node IDs. Do not run a broad command removing every orch.managed container on a shared daemon; that label can belong to somebody else's cluster. Preserve evidence of surviving objects and remove only your exact recorded IDs when authorized for the disposable lab. This is the same ownership principle used for process cleanup and database reset tests.

The upstream Compose commands above are source-checked expected steps until a matching execution record establishes their results. A successful package build or focused fake-runtime fixture does not count as a Compose run. If fixed ports or Docker are unavailable, keep the local validation and source trace, and use an owned ephemeral-listener/fake-runtime experiment for API transitions. Label that alternative accurately rather than replacing the missing daemon observation with an invented transcript.

A real-Docker alternative was executed for this edition and is retained in `validation/intake/docker-complete-flow-log-guard.json` with its complete JSON stdout. It connects the ordinary server handler to a real

agent Runner and Docker adapter inside a guarded teaching harness. The wrapper requires a cached image and checks exact owned identities; it publishes no workload host ports. This is an observed component composition, not a claim that the upstream Compose commands above ran.

The observed sequence includes one accepted service-created event, an owned running nginx task, an HTTP request made with docker exec inside that captured container, and a log request traversing server, agent and Docker. The response matched the nginx welcome page, while the log response contained nginx output. Timestamped task snapshots then show active replicas changing one to two to one, followed by terminal deletion. Captured IDs and cleanup evidence allow those observations to be correlated without inventing an external public endpoint. The earlier failed guard attempt remains separate from the corrected passed run.

This alternative supplies runtime evidence for deployment, scaling, logs and removal while leaving upstream Compose startup as source-checked procedure. It also exposes the resource-limit propagation gap measured in Chapter 5. A passing demonstration must preserve both its working path and its limitation, rather than treating every declared specification field as enforced because nginx answered one request.

The demo's useful result is a correlated sequence: accepted service intention, identified tasks, worker reports, independently inspected containers, scale changes and owned cleanup. Its limits are equally concrete: default state is memory-backed, resource propagation has a known gap, and this small cluster has no demonstrated HA/fencing contract. These facts let Atlas continue into lifecycle reasoning with evidence instead of a screenshot of a green endpoint.

Exercises

1. Explain why dev-up.sh succeeding does not prove the worker is ready or migrations ran.
2. Why can the example scale to two replicas on one host when public port is zero, and what changes with one fixed public port?

3. The agent container is stopped before service deletion. Explain the cleanup problem and the observations needed before removing anything.
4. A teammate reports “PostgreSQL started, therefore this demo survives server process loss.” Identify the wrong inference and design a narrow observation that tests it.

Worked answers include the ownership and rejection criteria for each decision.

Chapter 4 – Domain Models and State Machines

Atlas API has a service identifier, one assigned task, and a running container. Those are three different objects. If an operator scales the service down while an agent is reporting a successful health check, which statement wins? Reading status as a single mutable string cannot answer that question. The domain model separates what the server wants from what the worker has observed, and its transition helpers constrain how their disagreement can resolve.

Our objective is to explain one replica completely enough to reject an impossible report. We will use the active `pkg/types` definitions, the actual memory control plane, and the lifecycle tests. The presence of a `Deployment` or `Event` struct tells us what data can be represented; it does not establish a durable transaction, delivery guarantee, or production deployment topology. Those need separate implementation evidence.

By the end, you should be able to distinguish intent from observation, trace a replica through valid states, and explain why deployment versions and transition owners matter when rejecting stale reports.

Service Intent and Per-Replica Tasks

A `Service` contains identity, namespace, a specification, lifecycle status, and deployment version. Its specification carries replica count and image along with ports, environment, resources, and policy. A `Task` has its own identity, service reference, optional node/container references, image, version, and execution fields. See the service contract in pinned types and task contract in pinned types. The links are immutable snapshots, rather than statements about another revision.

For Atlas, `replicas: 2` is service intent. It does not create a two-process task. The control plane represents two separate units of work, each with its own assigned node and actual status. A task's version connects it to a

service specification at a particular deployment version. Comparing only `service_id` loses that distinction during an update: old and new replicas can belong to the same service while intentionally running different images.

Service versions are recorded by the memory implementation when creation, update, or scale changes occur. The version store implementation keeps specifications indexed by version. This is implemented history within a process, not evidence of PostgreSQL recovery. The helper assigns a Go struct into the version map; it is not a general deep-copy or immutability mechanism for nested maps and slices. Treat version identity as a domain contract without inventing stronger alias protection. A new process cannot reconstruct maps merely because the map values have stable identifiers. Preserve that distinction whenever a diagram labels something a version store.

The service lifecycle is deliberately narrower than task execution. `active` means the service is eligible for desired-state reconciliation; it does not mean all replicas are healthy. `deleting` records an intention to remove work. `deleted` is terminal for that service identity. The service transition validator permits active to deleting, then deleting to deleted. It rejects an immediate active-to-deleted transition, which would erase the explicit convergence phase.

This also explains why a deleted Atlas service should be replaced by a newly created service rather than revived by overwriting its status. Identity carries history. Reusing an old identity can confuse delayed reports, task ownership, and operator interpretation even if its display name looks convenient. The validator's contract is stronger than that convenience: deleted-to-active is invalid.

Desired Status Versus Actual Status

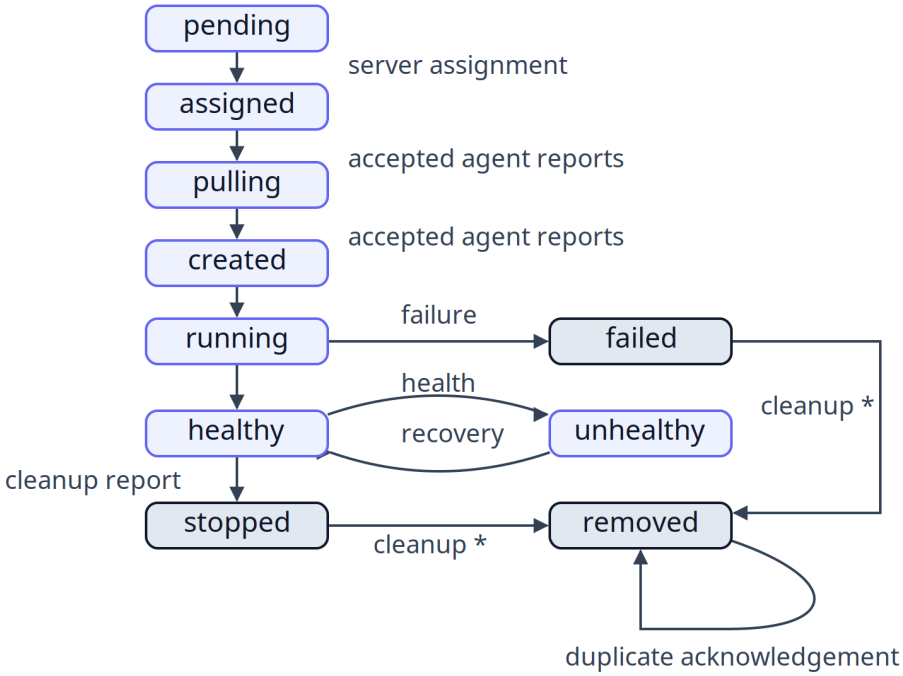
The following observations can all be legitimate moments in one replica's life. They describe a trace through permitted states, not a promise that every polling client will see every intermediate row.

Moment	Server-owned desired status	Actual status	Meaning
Work waiting for placement	running	pending	The requested execution lacks an assignment
Node chosen	running	assigned	A server assignment exists; no process is implied
Worker creates container	running	created	A runtime object exists without a running guarantee
Process observed	running	running	Runtime execution was reported
Health succeeds	running	healthy	A health report accepted at this point
Scale-down requested	removed	healthy	Last observation lags the new intention
Cleanup acknowledged	removed	removed	Task cleanup reached a terminal state

The literal JSON fields are `desired_status` and `actual_status`; neither is interchangeable with service `status`. The complete set of task status constants includes `pulling`, `starting`, `unhealthy`, `stopping`, `stopped`, and `failed` in addition to the table's values. `starting` and `stopping` are reserved progress states in the contract. Their presence does not mean the current agent emits them.

Agent report authorization is a second check beyond membership in the enum. The allowed report list excludes `pending`, `assigned`, `starting`, and `stopping`. The server owns assignment. Agents can report pulling, created, running, healthy, unhealthy, failed, stopped, and removed. This prevents an ordinary worker report from selecting a node or inventing a scheduling transition.

Here is a compact selection of legal actual-state transitions, with ownership written on the arrows. It intentionally omits many legal shortcuts. It is not a complete graph and should not be used to reject an arrow solely because that arrow is absent here.



* Cleanup to removed depends on desired-state checks.
Gray states are terminal for execution.
Selected legal edges; other valid transitions are omitted.

The static diagram has a high-resolution PNG companion at examples/diagrams/task-lifecycle.png for publishing targets that require raster artwork. Gray states are terminal for execution; cleanup to removed remains conditional on desired-state checks.

All names come from the actual-state validator and its permitted branches. `stopped`, `failed`, and `removed` are terminal for execution. The first two permit a cleanup transition to removed in the generic validator; removed has no route back to execution. A health failure is different: unhealthy can recover to healthy or running while failed cannot return to running. Replacement after failure needs a different task identity under the service's restart policy.

Desired status is even more constrained. Its normal progression is running to stopped or removed, then stopped to removed. Same-state updates are allowed. An actual task in failed does not justify changing desired status to failed: server intent and worker observation have different state machines even though their Go field types share `TaskStatus`.

Deployment Versions and Transition Owners

A deployment records from-version, to-version, strategy, status, and rollout limits. It describes an operation over service versions. It is not an alternative container identity. Atlas can therefore have an active service, a running deployment, and both old healthy tasks and new assigned tasks simultaneously. These states answer different questions and should remain separate in dashboards and scripts.

The deployment validator distinguishes pending, running, paused, rolling_back, succeeded, failed, and rolled_back. Succeeded, failed, and rolled_back are terminal. A rollback creates another deployment operation rather than mutating the terminal result of an earlier deployment into success. `paused` being a valid state does not establish an operator-facing pause command; representability and exposed control are separate integration questions.

The same caution applies to node status. Ready, draining, offline, and unknown describe placement eligibility and current observation, not a consensus protocol. Ready can become draining or offline; offline can become ready. A heartbeat timeout can justify recording offline, but it cannot prove that an isolated Docker host has stopped running its containers. The node type has no magical fencing effect.

Events add another record of an occurrence. Their type includes severity, source, related-object identifiers, details, and timestamp. Correlate an Atlas task transition with its task identifier and service identifier rather than reading the event message as a unique key. An event struct does not establish an exactly-once stream. Inspect the mutation and event-emission sites before asserting that duplicate reports cannot duplicate events.

The model developed in two distinguishable steps. The original core-model commit introduced richer types and validation. The lifecycle freeze later centralized transitions and changed API, scheduler, rollout, store, and control-plane callers. The exported patches show these were cross-cutting changes. Original prompts 02 and 29 explain intent, but their numbering is not proof of a one-prompt-one-commit execution history.

There are also older small wrappers under `internal/task` and `internal/node`. They are useful historical context, but they do not supersede `pkg/types`. In particular, using the older node wrapper as the authoritative enum would miss offline. Always trace which type the active API and agent actually serialize.

Rejecting an Impossible Transition

The smallest useful lab requires the locked Go toolchain and disposable application copy, without Docker or PostgreSQL. Start from unchanged source and run the actual transition tests:

```
go test -count=1 -v ./pkg/types -run 'TestValidate.*Transition|
TestAgentCanReportTaskStatus'
```

Our recorded run is `validation/labs/ch04/lifecycle-tests.json`; the exact Go binary, environment keys, UTC times, working directory, exit code, and stdout are retained. It passed. This observation covers the selected unit tests, including rejected pending-to-running and failed-to-running transitions. It does not demonstrate distributed ordering of two real agents or persistence across a process restart.

The domain error is explicit:

```
func invalidTransition(kind string, from any, to any) error {
        return fmt.Errorf("%w: %s %q -> %q", ErrInvalidTransition,
kind, from, to)
}
```

Exact source: pkg/types/lifecycle.go:320–322.

Callers can use `errors.Is(err, types.ErrInvalidTransition)` while retaining a diagnostic that includes the object kind and attempted states. Matching the full formatted message would couple an API client to human wording. Conversely, swallowing the error and updating the task anyway would destroy the contract the helper exists to enforce.

Consider the scale-down failure window. Atlas task actual status is healthy. The server sets desired status to removed. An old healthy report arrives before the worker receives the new intent. AgentCanReportTaskStatus rejects this report because cleanup intent permits only stopped, removed, or failed. A dashboard that blindly takes the last message would revive the replica; the desired-state check prevents that specific inconsistency.

There is a subtle boundary worth testing mentally. The generic validator allows failed-to-removed cleanup, but an agent report also depends on desired state. With desired running and actual failed, the agent gate accepts only the duplicate failed status. With desired removed, a removed report is allowed. A generic transition diagram cannot replace the caller's additional conditions. This is why our diagram annotates terminal cleanup rather than declaring all such reports unconditionally accepted.

For a failure investigation, retain the task identity, desired status, actual status, incoming report, and which helper rejected it. Do not log environment secrets. A rejected report is successful enforcement of this contract, not automatically evidence of a broken agent. The next investigation is whether the agent was delayed, whether desired state changed, or whether a caller submitted an invalid report.

The lab modifies no baseline files, starts no daemon, and creates no containers. Cleanup consists only of optional removal of a separately created test cache; keep the evidence record. A repeated run should use another record filename rather than overwriting the first outcome. If the selected test fails on another version, stop and compare that version's helpers before treating these arrows as current.

Before moving to the API boundary, distinguish three forms of success. A transition helper returning nil proves a proposed local edge is legal. A mutation method completing proves that its own checks and updates completed in that implementation. An agent acknowledging removal proves the reported task reached cleanup, within the evidence available to that server. None alone proves an unreachable host has relinquished a volume, that a previous server transaction survived a crash, or that every observer has received an event. Atlas operators need each statement separately because each has a different failure window and a different owner.

Exercises

1. For a newly assigned Atlas task, list identity, intent, and observation fields needed to correlate a later container report. Explain why service name alone is insufficient.
2. Decide whether unhealthy-to-healthy, failed-to-running, and failed-to-removed are allowed by the generic actual-state validator. Then evaluate the final case through the agent gate with desired running and desired removed.
3. Explain why an active service with a succeeded deployment can still have a failed task. Identify which object a restart replacement changes.
4. An operator proposes accepting healthy reports after desired status becomes removed because they are more recent than the deletion request. Explain the unsafe assumption and give the correct rejection evidence.

See the worked answers. They distinguish the helper's local rules from delivery, persistence, and host-fencing claims that require later experiments.

Chapter 5 – Service Specifications, Resources, and Ports

Atlas API now needs three replicas rather than one. The deployment author proposes 250 millicores, 128 mebibytes, and the same public port for every replica. Each value looks reasonable in isolation. Together they describe capacity requirements and a possible host conflict that validation alone cannot settle. This chapter follows those fields from YAML into domain values, then checks which downstream paths actually use them.

The distinction matters immediately: a parser can accept a correctly shaped request while a placement path ignores its resource requirements, and a Docker adapter can support limits while its caller omits them. We will calculate the intended budget, reject malformed manifests through the actual CLI parser, and keep admission, scheduling, and runtime enforcement as separate claims.

You will be able to parse CPU and memory units, calculate replica budgets, predict node-scoped host-port conflicts, and identify the boundary between admitted specifications and runtime enforcement.

From YAML to a Validated Service

The deployment file is a CLI-specific format. Its top-level `resources` mapping contains string CPU and memory quantities; `ports` entries contain `container` and `public`; placement labels are nested under `placement.labels`. These names do not match the domain JSON representation one for one. Inspect DeployFile and its nested types in the parser before copying an HTTP payload into YAML.

The parser reads bytes, decodes them with known fields enabled, validates the deployment representation, converts fields to domain types, validates the resulting specification, and applies defaults. ParseDeploy is the integration point. Known-fields decoding means a misspelling such as `replica` is an error rather than an ignored suggestion. This is valuable operator feedback: silently interpreting a typo as zero replicas would be a materially different deployment.

Our valid teaching fixture is stored in `examples/teaching-extensions/ch05/atlas-valid.yaml`. It is original book-side input, not an upstream fixture or a change to the baseline:

```
name: atlas-api
image: nginx:1.27
replicas: 3
ports:
  - container: 80
    public: 0
resources:
  cpu: 250m
  memory: 128Mi
placement:
  labels:
    role: worker
```

The parser maps this into both requests and limits. That equality is a behavior of the CLI deployment conversion, not a universal domain restriction. The domain can represent different requests and limits, provided validation accepts them. A direct HTTP client submitting the domain representation must therefore follow its own field names and validation rules.

The valid fixture is locally admissible. It does not prove a worker with `role=worker` exists, that the registry is reachable, or that three containers will fit on that worker. Those questions belong to later operations. Local validation also cannot check whether a referenced secret exists in the selected namespace or whether its operator may create a service there.

Four sibling fixtures deliberately replace one field. `atlas-invalid-cpu.yaml` uses `bananas`; the memory variant uses `1.5Mi`; the port variant uses container port 70000; the field variant misspells replicas. Their failures are distinct: quantity parsing, integer-suffix memory grammar, bounded port validation, and strict YAML field decoding. The retained actual responses make those distinctions observable instead of calling all invalid input a YAML error.

CPU and Memory in Exact Units

The internal resource representation uses CPU millicores and memory bytes. For Atlas, 250m is 250 millicores, or one quarter of a CPU. Three replicas request 750 millicores, equivalent to 0.75 CPU. Memory 128Mi equals 128 × 1,048,576 = 134,217,728 bytes per replica. Three replicas total 402,653,184 bytes, or 384Mi. These are arithmetic budgets derived from declarations, not measurements of live utilization.

Quantity	Per replica	Three replicas
Requested CPU	250 millicores	750 millicores
Requested memory	134,217,728 bytes	402,653,184 bytes
CLI-converted limits	Same values as requests	Same aggregate values

CPU parsing accepts a positive integer with `m`, or a positive finite decimal number of CPUs which it multiplies by 1,000. Memory parsing accepts positive integers with binary suffixes Ki, Mi, Gi; decimal suffixes K, M, G; or a plain byte count. Thus 128M means 128,000,000 bytes, which is less than 128Mi. Uppercase and lowercase spelling are part of this grammar, not stylistic choices.

The exact CPU implementation is short enough to examine directly:

```
func ParseCPU(value string) (int64, error) {
        value = strings.TrimSpace(value)
        if value == "" {
                return 0, nil
        }
        if strings.HasSuffix(value, "m") {
                raw := strings.TrimSpace(strings.TrimSuffix(value,
"m"))
                parsed, err := strconv.ParseInt(raw, 10, 64)
                if err != nil || parsed <= 0 {
                        return 0, fmt.Errorf("invalid CPU value %q",
value)
                }
                return parsed, nil
        }
        parsed, err := strconv.ParseFloat(value, 64)
        if err != nil || parsed <= 0 || math.IsNaN(parsed) ||
math.IsInf(parsed, 0) {
                return 0, fmt.Errorf("invalid CPU value %q", value)
        }
        return int64(parsed * 1000), nil
}
```

Exact source: pkg/types/types.go:898–916.

The HTTP resource decoder also distinguishes JSON numbers from strings: numeric CPU is already millicores, while a CPU string such as "1" passes through ParseCPU and means 1,000 millicores. A raw numeric 0 is rejected, whereas an absent field represents omission. Therefore copying a Go zero-valued struct into raw JSON is not equivalent to omitting resource fields.

An empty string returns zero so normalization can supply defaults. Explicit `0m` is rejected rather than interpreted as unspecified. Decimal CPU is converted with a floating-point multiplication followed by an integer conversion. Do not infer arbitrary precision or rounding guarantees from this small parser. Extremely small positive decimals can truncate to zero before defaulting, and arithmetic at extreme integer magnitudes is not comprehensively guarded here. For the lab we use ordinary, unambiguous quantities and retain those boundary questions for a robustness review.

The default resource requirements are 100 millicores and 128Mi for both requests and limits. WithDefaults substitutes zeros component by component, then applies strict validation. Requests cannot exceed a positive corresponding limit. Distinguish raw nonnegative validation from normalized strictly positive validation: the stages intentionally accept different representations of omission.

Suppose a direct API caller requests 250 millicores but leaves its CPU limit zero. Defaulting the limit to 100 would produce a request above the limit and fail strict validation. The CLI avoids that particular ambiguity by assigning parsed resources to both sides. Knowing where defaults apply prevents a seemingly similar YAML and HTTP example from silently teaching different behavior.

Labels, Replicas, and Finite Host Ports

Placement labels restrict eligible nodes; they do not increase capacity. Three Atlas replicas constrained to one worker still share that worker's CPU, memory, and network bindings. A request for fixed public TCP port 8088 on every replica asks for the same host binding repeatedly. On one node, that is a conflict even though every individual port is within the allowed range.

Host port accounting is node-scoped and protocol-aware. Another worker may bind the same numeric TCP port on its own host. TCP and UDP with the same number are distinct keys in the scheduler's reservation model. A public port is not a globally unique cluster service address, and changing replica count does not automatically provide a reverse proxy or load balancer.

The scheduler package's allocatePorts implementation checks explicit ports against existing and newly planned reservations. For dynamic requests it scans the bounded range 30000–32767. Under that planner, two replicas with fixed TCP 8088 cannot both be planned on the same worker. A second eligible worker may allow placement; otherwise a task remains unplaced. Valid syntax and schedulability are different outcomes.

Here is the important active-path qualification. The memory control plane immediately creates assigned tasks using ready-node round robin. It invokes assignPortsForNodeLocked instead of that planner. The helper allocates a free dynamic port but does not reject an already-reserved explicit port. Therefore the planner's collision tests do not prove the immediate creation path prevents a fixed-port collision. A container creation error may expose the conflict later at Docker.

This is a source-confirmed integration gap recorded as DOC-014 in the contradiction ledger, not permission to improve the upstream implementation inside this book. Do not demonstrate it by binding arbitrary ports on a shared host. The safe teaching input uses public zero, and any experiment with fixed bindings needs disposable hosts, captured ownership, and explicit cleanup.

Even dynamic reservation has a boundary: the orchestrator's in-memory view does not know every unrelated host process. A number free in its task maps could already be occupied outside orch. Runtime creation remains a separate enforcement point. Successful allocation is a statement about the planner's known state, not a kernel reservation held before the Docker operation.

When Admission Is Not Runtime Enforcement

Resources have a similarly revealing path. They are present in ServiceSpec and are used for admission/accounting. The Docker adapter supports memory bytes and NanoCPUs in HostConfig. Yet the server's AgentTask envelope does not contain resources, and the agent's ContainerSpec construction does not assign them. Follow the envelope definition, agent creation fields, and Docker HostConfig conversion together before asserting a live limit.

This gap was observed with real Docker. The guarded record `validation/intake/docker-resource-boundary-corrected.json` retains the exact command and scope. A service requesting limits of 250m and 128Mi reached Docker through the ordinary handler and agent Runner with inspected Memory=0 and NanoCPUs=0. A separate direct adapter call

with supplied limits produced Memory=134217728 and NanoCPUs=250000000. The direct result demonstrates adapter capability; it does not repair the service path.

The harness used cached images, exact owned container IDs, and no host workload ports. Containers were removed and cleanup recorded. It was a teaching harness around real upstream components, not the upstream Compose deployment. The initial corrected-attempt history remains available rather than being replaced with a fabricated clean transcript.

Our local parser lab is independently reproducible without Docker. With the disposable baseline's built CLI and Python 3 available, run:

```
python3 /home/popac/alekpopovic.github.io/books/inside-orch/
examples/teaching-extensions/ch05/check_manifests.py /tmp/inside-
orch-labs-cbfc94ade5b5/bin/orch
```

The harness invokes the actual `orch validate` command on each checked-in fixture. It asserts valid input exits zero and each invalid input exits nonzero with its expected category of diagnostic. The record `validation/labs/ch05/parser-manifests.json` passed, with each child command, exit code, stdout, and stderr retained. The harness's success means the rejection assertions passed; it does not mean invalid manifests were accepted.

A useful failure window is between accepted intent and worker container creation. The server can count requests toward quota while the agent has not yet created a container, or while the eventual runtime limits are absent. Operators must therefore compare requested capacity with inspected enforcement rather than assuming accounting is usage or isolation. An admission success does not establish that a noisy replica is constrained.

History helps explain the separate layers. Resource accounting, host-port allocation awareness, and service validation changed different packages. Their existence does not make all paths converge automatically. Original prompts 17,34,43 are intent records; current callers and actual observations decide the capability label.

No parser run changes application state. Keep the evidence and fixtures; there is no database or container cleanup for this local lab. The Docker experiment has its own explicit container cleanup evidence and must not be generalized into permission to remove unrelated host resources.

For the Atlas capacity review, retain three columns: declared request, placement decision, and runtime inspection. A declaration gives a reproducible budget; a placement decision names the actual node and its known reservations; inspection shows what Docker received. A disagreement between columns is actionable evidence. Changing a YAML value repeatedly until deployment succeeds obscures the cause, because it mixes parser rejection, scheduler eligibility, host binding, and adapter propagation into one symptom. Reproduce the smallest failing boundary first, then decide whether the implementation or the deployment requirements need a separately reviewed change.

Exercises

1. Calculate CPU millicores and memory bytes for five replicas requesting 0.4 CPU and 64Mi each. Explain why 64M yields another total.
2. Explain the different rejection points for `cpu: bananas`, `memory: 1.5Mi`, and a misspelled replicas key. Run the fixture harness and identify each actual response.
3. Can two replicas publish TCP 8088 on one worker? Compare the scheduler planner's answer with the immediate memory assignment path.
4. A reviewer sees correct Docker adapter tests and claims Atlas has CPU isolation. Trace the missing fields and state which actual observation would be necessary to support that claim.

See the worked answers.

Chapter 6 – Follow One Request Through the Control Plane

Atlas API's deployment has passed local parsing, but the interesting architectural question starts when the server accepts it. Which object changes first? Which controller observes that change? What survives if the server process exits after returning success? This chapter follows one create request through the executable composition, then measures the last question with a fresh binary process and one precisely owned Docker container.

Keep the previous chapter's resource and port findings nearby. They are examples of why an architecture diagram must follow callers, not just package names. A scheduler, database store, and runtime adapter can all exist while the ordinary service path uses only some of their behavior.

You will be able to trace the active constructor and mutation call sites, distinguish immediate assignment from periodic controllers, and classify a measured process restart without calling it database recovery.

Server Assembly and Active State Ownership

The decisive constructor is in the server entry point:

```
        envelope, err := secrets.NewLocalEnvelope(cfg.SecretKey)
        if err != nil {
                return err
        }
        controlPlane :=
controlplane.NewMemoryService(controlplane.WithSecretEnvelope(envelo
pe), controlplane.WithClusterPolicy(cfg.ClusterPolicy),
controlplane.WithRetentionConfig(cfg.Retention))

controlPlane.SetNotificationDispatcher(notifications.NewDispatcher(c
ontrolPlane, nil))
        serverMetrics := metrics.NewServer()
```

Exact source: cmd/orch-server/main.go:142–148.

A local secret envelope is created, then passed into a new MemoryService along with cluster policy and retention configuration. The active state owner is this particular in-memory object. The database URL in configuration does not change that constructor. Migration subcommands and optional schema checks can use PostgreSQL without turning service maps into a PostgreSQL-backed control plane.

The entry point then constructs GitOps and cron controllers, a scheduler controller, a rollout controller, and a node monitor around this object. Notification dispatch is configured on it. Optional DNS uses it as its source. Usage snapshots and retention pruning are also driven by loops. These are real integrations because their constructors and Run or RunOnce calls are present in the active startup path of main.go.

Timing differs between those loops. Scheduler RunOnce is called on a one-second ticker. Rollout Run runs with a five-second interval. Usage is captured immediately and on a minute ticker; retention uses a daily ticker. Node-monitor timing comes from configuration. A graph with one generic arrow labelled controller loop would conceal these different observation windows and owners.

Some implemented packages are absent from this assembly. The standalone reconciler is not constructed here. Leadership leases do not gate these loops. Autoscaling policy can be represented and its package tested, but the autoscaler is not started by this binary. PostgreSQL store methods remain separately usable implementations. These classifications describe the locked source, not a suggestion that the missing connections can be inferred from configuration names.

The HTTP server receives the same MemoryService pointer, plus authentication options, metrics, and the GitOps syncer. Its graceful shutdown closes the listener and cancels loops through the process context. That is useful process hygiene. It is not a transaction flush of the control plane's maps to a persistent store.

Mutation Paths and Controller Loops

The create-service handler decodes a request containing `spec`, performs the boundary checks, and invokes CreateService. Inside MemoryService.CreateService the sequence is specific: check context cancellation, resolve image representation, validate the specification, run admission, normalize resource defaults, validate again, then acquire the mutation mutex.

The resolver's name should not hide its actual default behavior. It parses image references rather than fetching registry manifests. Setting pinning policy does not manufacture a remotely verified digest. For Atlas, keep requested image, supplied digest, and remotely resolved identity separate until a resolver implementation and an actual registry observation support the stronger claim.

Under the mutex, the method checks namespace existence, duplicate names, referenced secrets, registry credentials, and quota. It constructs an active service at deployment version 1, stores its specification version, reconciles its replica tasks immediately, appends a service-created event, and returns. The critical section coordinates these changes within that process. It is not a database transaction spanning a durable event log and a future Docker operation.

Boundary	Owner	What completion establishes
YAML parsing	CLI	A domain specification could be constructed locally
CreateService mutation	MemoryService	Accepted intent and related map updates in this process
Immediate assignment	MemoryService	Assigned task identity and selected ready node
Agent container operation	Agent Runner and Docker adapter	Runtime operation outcome for that task
Accepted status report	API and MemoryService	Updated observed task state under report rules

Notice that creation can assign tasks before the periodic scheduler tick. Immediate reconciliation chooses a ready eligible node using round robin and creates an actual status of assigned. It does not submit every new replica to scheduler.Plan. This is why the planner's resource and explicit-port checks cannot be promoted to blanket guarantees for ordinary creation.

The scheduler loop still exists. Its presence is not fictitious, but its inputs and applicability must be traced. A controller that operates on pending tasks cannot retroactively prove fit for tasks already assigned by another path. Architecture review should ask which mutation creates each input state, then which controller consumes it.

The worker subsequently lists assigned tasks. ListAssignedTasks builds an envelope from the task and its service, resolving environment values and registry credentials, and adding ports, healthcheck, and security context. That envelope omits resources. The agent then builds ContainerSpec without resources. The previous chapter's actual Docker inspection therefore follows from a specific data-flow break rather than a speculation about Docker's limit support.

A status report returns through the API, checks agent credentials and task ownership, and enters ReportTaskStatus. The memory implementation validates the node relationship and the desired/actual state gate before updating container ID, status, timestamps, and failure information. A duplicate terminal report returns early. This is a caller-specific idempotency behavior; it does not establish universal exactly-once event delivery.

The MemoryService Restart Experiment

The failure window is between accepted intent and the next fresh server process. Docker is a separate daemon. Ending the control-plane process does not inherently terminate its containers, while recreating MemoryService does recreate empty maps. We want to observe those two facts in one bounded experiment without confusing a controller restart with process replacement.

The book-side harness is `examples/teaching-extensions/ch06/observe_container_restart.py`. It verifies every locked source file in the disposable working copy before launching the binary. Prerequisites are the already built baseline server, Python 3, access to Docker, and a cached `nginx:1.27-alpine` image. It refuses an existing evidence output and uses a newly selected loopback port. No pre-existing server endpoint is contacted.

The harness creates one container from the cached image with network disabled, no published ports, and no host mounts. It retains the exact returned container ID. This container is created manually by the harness, not by an upstream Runner. After registering a synthetic node and creating a service through the first actual server binary, it authenticates a running task report containing that container ID. The server accepts the relationship; the evidence explicitly labels it manually reported rather than automatically created or adopted.

Run the exact illustrated procedure from the website root with permission to access the local Docker daemon:

```
python3 books/inside-orch/examples/teaching-extensions/ch06/
observe_container_restart.py \
  --working-copy /tmp/inside-orch-labs-cbfc94ade5b5/orch \
  --server /tmp/inside-orch-labs-cbfc94ade5b5/bin/orch-server \
  --output validation/labs/ch06/combined-restart-repeat.json
```

Use a new output name on each run. The recorded execution used `combined-restart.json`, with its command wrapper in `validation/labs/ch06/combined-restart-command.json`. The repeat path above is intentionally not claimed as an executed second run. The script contains all exact Docker and HTTP calls, bounded startup polling, process termination, and cleanup rather than requiring the reader to reconstruct them from a transcript.

The actual run passed. Before shutdown the first binary listed a service, its assigned-and-reported-running task, and the registered node. After a clean stop and a fresh server process on the same origin, all three lists were empty. The manually created container with the captured task

container ID was still running. The custom namespace also disappeared, while the default namespace was recreated. Both process IDs and UTC observations are in the record.

That observation demonstrates loss of active process memory while an independently managed runtime object survives. It does not demonstrate database recovery, agent reconnection, automatic container adoption, or safe recreation of an identical service. In particular, a newly created service after restart would have a different identity. We intentionally stop before asking another controller to create competing work.

Cleanup is exact and independently checked: the harness terminates only the server processes it launched, removes only its captured container ID, and verifies that inspecting that ID subsequently fails. It does not remove all containers with an orch label, prune Docker, or delete database schemas. Evidence survives cleanup. If the container ID cannot be captured or startup fails, the failure must remain in the record rather than being converted into an expected success.

Integration Gaps as Architectural Evidence

The restart result is useful because it answers a narrow question conclusively. A controller test that recreates a rollout controller around the same MemoryService answers another question: can that controller resume from still-existing state? Both can pass while a new server process loses its maps. The original chaos test named server restart actually retains the same memory service; DOC-007 preserves that distinction in the ledger.

Likewise, restoring PostgreSQL tables would help an implementation that reads those tables as its active state. The default binary does not. DOC-008 records the mismatch between a backup runbook's assumed composition and the constructor we inspected. Merely setting a database URL after a crash cannot rehydrate service, task, credential, and event maps that were never stored there.

The separate real-Docker flow record `validation/intake/docker-complete-flow-log-guard.json` demonstrates ordinary handler, Runner, and adapter interactions: deployment, actual nginx HTTP from inside the owned container, logs through the server-agent path, scale1→2→1, events, and deletion. It uses a guarded teaching composition rather than upstream Compose. Combining that record with our restart result is valuable, but their distinct scopes must remain visible.

A practical architecture review can now name concrete change requirements. A durable composition would need complete store parity for mutations and reads, state reconstruction, transaction boundaries for quota and assignment, credential continuity, controller restart behavior, and a policy for surviving runtime objects. Leadership would also need to gate each relevant controller consistently rather than only adding a lease package. These are future integration requirements, not repairs made to the locked source.

The implementation history reinforces that caution. The REST API commit introduced the early executable control plane. The later operations batch added many modules and hardening changes. Reading its patch shows broad additions, not a proof that every design document became a connected production subsystem. Original prompts 04,10,25 remain development evidence; current startup and measured failure boundaries govern this edition.

For Atlas, the immediate operational decision is straightforward: treat the default binary as a development control plane with process-local intent. Preserve the actual limitations when teaching its useful packages. A precise integration gap is more informative than either calling the whole project production-ready or dismissing its independently implemented behavior.

Exercises

1. Identify five active background activities in the server entry point and distinguish them from immediate service assignment.

2. Trace where an Atlas create request acquires the mutation mutex and which checks precede or follow it. What atomicity does that mutex provide?
3. Explain why the combined restart observation does not establish automatic adoption of a surviving Docker container. Name the manual step that limits the claim.
4. A proposed recovery plan changes only DATABASE_URL and restarts the binary. Use the constructor and observed state to explain why this plan is insufficient.

See the worked answers.

Chapter 7 – HTTP Contracts and Compatible APIs

An Atlas deployment script should distinguish a malformed service request from a missing service, insufficient authorization, and a temporarily unavailable server. Parsing whatever human message happened to arrive is fragile. The public API supplies structured errors, versioned routes, request identifiers, and a hand-written Go client; this chapter tests how those pieces agree and where the documentation remains incomplete.

We will keep two boundaries separate. Public user requests express workload intent. Internal agent requests register workers and report execution. Sharing an HTTP listener does not give those contracts the same credentials, audience, or documentation policy. The public OpenAPI document deliberately excludes agent paths.

You will be able to follow authentication and validation, interpret structured errors, and compare the public client and OpenAPI against actual routes without merging public and agent contracts.

Request Boundaries and Middleware Order

The handler assembly is unusually compact:

```
func (s *Server) middleware(next http.Handler) http.Handler {
        return
requestIDMiddleware(apiVersionMiddleware(s.accessLogMiddleware(s.rec
overyMiddleware(s.timeoutMiddleware(s.userAuthMiddleware(next))))))
}
```

Exact source: internal/api/api.go:1255–1257.

Read the nested calls from the outside inward: request ID, API version, access log, panic recovery, timeout, user authentication, then route dispatch. Consequently rejected authentication can still receive a

request ID and version header. The recovery layer is inside access logging, allowing recovered failures to be reflected in the outer request observation. Timeout behavior remains separate from proof that a domain mutation was rolled back.

Namespace selection occurs in user-auth middleware before role evaluation. `X-Orch-Namespace` takes precedence over the query parameter, with normalization to default when omitted. The namespace is validated and placed into context, then the required role is looked up for the route. When no user JWT secret is configured, the middleware bypasses user token enforcement. A bearerAuth entry in OpenAPI therefore documents the configured public contract; it does not prove the development server currently requires tokens.

With JWT configured, the middleware parses a bearer token, validates it, applies static-user overrides when configured, selects the namespace role, and checks whether that role satisfies the route's required role. Static overrides clear token namespace-role assignments. These checks decide whether the handler is reached. They do not validate a Docker operation or reserve resources on a worker.

A create request then reaches createService with a spec envelope. The decoder rejects unknown JSON fields and requires the body to decode successfully. The handler validates the specification and delegates mutation to the control plane. The preceding chapter traced admission, normalization, references, quota, and map mutation. A successful response is HTTP 201 with a top-level `service` object, not the bare specification.

The distinction between unknown request and response fields is intentional. The server rejects an unknown request field to catch mistakes at the command boundary. The public Go client's ordinary JSON decoder ignores unknown response fields so an older client can survive additive server information. Applying strictness in both directions would make harmless response additions break existing callers.

Domain Errors as Public JSON Contracts

The API maps domain conditions into stable error codes and HTTP statuses. CodeOf recognizes missing objects, duplicate/conflicting objects, invalid state, and cancellation/deadline failures. Admission and quota receive additional translation with structured details. Unexpected internal errors expose a generic message rather than the raw internal failure string.

Condition	HTTP status	Public code
Invalid UUID or invalid specification	400	invalid_argument
Missing object	404	not_found
Duplicate service name	409	conflict
Quota exceeded	409	quota_exceeded
Missing configured user credential	401	unauthorized
Insufficient configured role	403	forbidden

The actual response envelope contains `error`, whose fields include `code`, `message`, optional `request_id`, and optional `details`. These are defined in the API DTO and response writer. The following is an illustrative shape, not copied terminal output:

```
{"error":{"code":"invalid_argument","message":"service id must be a
UUID","request_id":"example-only"}}
```

Clients should branch on the stable code and status while preserving the request identifier for investigation. The message explains the issue to a human and can change wording. A request ID is correlation metadata, not a secret, authentication proof, or transaction commit identifier. The middleware accepts a supplied identifier or generates one; it does not prove an end-to-end distributed trace exists.

The public client decodes non-2xx responses into `client.APIError`, carrying status, code, message, request ID, and details. decodeAPIError also falls back to an error containing status when the body is not the

expected JSON. This distinction matters behind a proxy: an HTML gateway failure should remain an HTTP failure, not be misreported as an absent service.

Redaction belongs at multiple boundaries. App-error details recursively redact sensitive keys and URL user information according to the implemented policy. That mechanism does not authorize logging an entire request body, token, or resolved secret environment. Our smoke tests use synthetic values and discard server log output. They assert the presence of correlation metadata without printing credentials.

The v1 Compatibility Surface

Public workload routes are registered under `/v1`, while operational probes remain unversioned. The version middleware sets `API-Version: v1` on v1 responses. `/v1/version` exposes API, server, agent-range, and schema metadata. This is useful compatibility information, but an advertised schema range does not establish durable store integration.

The repository documents additive response fields and optional request fields as compatible changes. Renaming a field, changing its meaning, requiring previously optional input, or changing stable error codes requires a new contract version. Adding an enum value also requires clients that tolerate unknown values; code that exhaustively panics on a new task status can still break even when the JSON structure remains valid.

The deprecation wrapper can set Deprecation, Sunset, and a migration Link header. Its implementation is available for routes that need it; the presence of the helper does not mean every current endpoint is deprecated. No v2 route is registered. Our actual test checks that `/v2/services` returns 404 rather than assuming a future version exists because the guide describes how one should be introduced.

Compatibility is easiest to demonstrate with a small controlled response. The teaching smoke test presents the current client with an added `future_hint` top-level field. The existing service name still decodes.

Another fixture renames the nested spec name to `service_name`; decoding itself succeeds, but the client's expected `Name` value becomes empty. That silent loss is precisely why a renamed field is breaking even when no JSON syntax error occurs.

These fixtures are deliberately synthetic server responses. They demonstrate the pinned client's decoding behavior, not that the real API already emits future_hint or service_name. The real-handler part of the same test remains separate and creates Atlas through the actual API.

OpenAPI and the Client Agree—or Differ

The public client is hand-written. Its methods construct paths, wrap request bodies, decode response envelopes, and attach bearer and namespace headers. CreateService builds its request explicitly; there is no evidence here of a generator transforming OpenAPI into this implementation. Original development prompt 61 requested generated clients, but the current artifact and workflow govern what we can claim.

There is a concrete client-versus-raw-HTTP distinction at resource defaulting. Public CreateService normalizes a zero-valued specification before serialization. Directly json.Marshal-ing that same raw Go specification emits numeric CPU 0 and memory 0, which the resource decoder rejects. Omitted fields or empty request/limit objects permit defaulting. Our expanded smoke test observed raw-zero HTTP 400 followed by successful public-client creation using the same original Go value. DOC-016 records the mismatch; validation/labs/ch07/public-contract-raw-zero.json preserves the actual run. This is a boundary of the current DTO/conversion path, not a baseline repair.

The upstream OpenAPI test parses YAML, checks nonempty top-level sections, requires a selected list of paths, and rejects internal agent paths. It does not perform full OpenAPI semantic validation, verify every response schema, or compare all handler methods automatically. Passing that test is useful evidence of its stated checks, not full contract conformance.

We therefore add a narrow, reproducible static comparison in `examples/teaching-extensions/ch07/compare_routes.py`. It extracts registered HTTP method/path pairs from the pinned router, normalizes path-parameter names, and compares them with parsed OpenAPI operations. Agent routes are classified as intentionally internal; metrics is treated as operational. This is still not schema validation or live response checking.

The actual comparison passed its expected-gap assertions and found three public GET operations missing from OpenAPI: `/v1/discovery/services`, `/v1/discovery/services/{name}`, and `/v1/integrations/traefik/config`. Their registrations are visible together in the router. DOC-015 records the omission, with `validation/labs/ch07/openapi-routes.json` retaining the command and output. We leave the immutable upstream document unchanged.

An omission has concrete consequences. A consumer building documentation or a client from only the schema would miss those discovery/integration operations. That is different from a wrong existing response schema, which can cause a generated consumer to misread data. Keep method coverage, schema correctness, authentication description, and actual behavior as separate review dimensions.

Atlas Contract Lab

The canonical extension is `examples/teaching-extensions/ch07/contract_test.go`. Copy it unchanged into `.book-labs/ch07/contract_test.go` in the disposable source tree so Go's internal-package rules permit the real handler imports. Use the locked Go toolchain and its matching GOROOT/module cache. The actual application working directory must be the disposable copy, never the website root or immutable evidence directory.

```
go test -count=1 -v ./.book-labs/ch07
```

The observed command record is `validation/labs/ch07/public-contract-loopback.json`. It passed with ephemeral loopback test servers. The first attempt, retained separately, failed because the

sandbox denied listening on a local socket; that was an environment limitation, not evidence that the API contract failed. The authorized retry allowed only the local test execution and required no Docker, database, or external service.

The real-handler test creates Atlas, checks its returned identity/name, requests a non-UUID service ID, and verifies a 400 invalid_argument APIError with a request ID. It submits an unknown top-level request field and expects 400 with the v1 header. It also verifies absent credentials produce 401 when JWT enforcement is configured, and v2 remains 404. The additive/renamed fixtures test a different, explicitly controlled decoding boundary.

A useful failure window remains after the server mutates state but before a response reaches the client. A transport error is not proof that creation failed. The caller should inspect identity and name under the intended namespace before deciding whether another create is appropriate. Duplicate-name conflict can help identify an already accepted request, but it is not a universal idempotency key. The current API contract does not provide a durable exactly-once create guarantee.

The harness starts only ephemeral servers and closes each with defer. It modifies no baseline source and creates no containers or tables. Retain the extension's byte hash alongside its copied version, and preserve failed and passed records separately. Reproducing a test means running that copy against the specified source, not replaying a previous transcript.

History includes structured errors, OpenAPI and client additions, and policy/namespace integration. Inspecting those patches shows separate contract evolution. It does not make the schema a complete executable substitute for current handlers, or transform a hand-written client into generated output.

For Atlas consumers, the operational checklist is therefore concrete: retain stable response envelopes, tolerate additive response fields, validate required values after decoding, classify structured errors, and

keep the selected namespace visible in automation configuration. When an error body is absent, preserve transport or HTTP failure rather than inventing a domain cause. When schema and router differ, record the exact method and revision so a future edition can review a real correction. This workflow makes compatibility review a set of observable checks instead of a claim inferred from a version label.

Exercises

1. Explain why a rejected JWT can still have an API-Version header and request ID. Trace the actual middleware nesting.
2. What should an operator script branch on when a service ID is malformed? Explain the roles of code, status, message, and request ID.
3. Reproduce the additive-response fixture and explain why the renamed-field fixture is breaking despite successful JSON decoding.
4. A reviewer calls every agent route missing from public OpenAPI a defect. Correct the assumption, then name the three genuine public method omissions found by this lab.

See the worked answers.

Chapter 8 – A CLI That Operators Can Automate

The Atlas operator wants a simple answer in a deployment script: does the server currently report the desired replicas available? A table meant for a terminal is a poor input format for that decision. The pinned CLI provides JSON output, predictable configuration precedence, and distinct local validation, but an automation script still needs to check the correct fields and preserve failures in its exit status.

We will build a small status checker as a book-side teaching extension. It calls the real CLI rather than guessing HTTP payloads, requires an explicit server address, and describes its result as server observation rather than a fresh host-health guarantee. A reproducible lab executes the same commands against an owned fresh server with synthetic configuration.

You will be able to predict configuration precedence, distinguish local validation from server acceptance, and automate current JSON output while retaining actionable failures and explicit scope.

Cobra Composition and Local Validation

The root command is assembled with injected output writers and a client factory. This makes command behavior testable without hard-wiring a network implementation into every subcommand. NewRootCommand registers version, validation, node, service, deployment, scaling, events, logs, namespace, and later feature groups. An available command group establishes CLI wiring, not a claim that every related package has complete end-to-end integration.

Persistent flags include `--server`, `--token`, `--namespace` with `-n`, `--output`, and `--config`. Output is restricted to table or json in the root pre-run check. Do not invent `--format`, `--wait`, or a default server flag because another orchestrator uses them. Read each current subcommand before illustrating an operator procedure.

The `validate <file.yaml>` command invokes ParseDeployFile directly, prints a local validity message, and returns its error. It does not construct the API client. Consequently a missing server URL does not prevent local deployment validation. The chapter 5 harness actually exercised that property while rejecting invalid fixtures.

`deploy <file.yaml>` uses the same parser but then constructs a client and submits the specification. Local parsing catches shape and quantity errors. Server admission can still reject namespace, authorization, policy, references, or quota. A successful validate followed by a failed deploy is not inconsistent: the commands cover different boundaries.

Service references can be names or UUIDs. resolveService uses GetService for a UUID and otherwise lists services and selects a matching spec name. A name-based `service inspect atlas-api` therefore has a list-and-select step, while an ID points directly at an identity. During deletion and recreation, reusing the same name can select a different service than a previously captured ID.

The CLI maintains its own APIClient wrapper and structured APIError formatting. The public client package contributes shared URL normalization, but the CLI is not simply an alias for every public-client method. Follow the injected client interface and actual implementation when adding an automation command; the existence of a similarly named method elsewhere does not connect it automatically.

Configuration Precedence You Can Predict

CLI configuration begins as an empty configuration, optionally reads YAML, then applies nonempty environment values, then nonempty explicit overrides. The server, token, and namespace use the same order. Namespace falls back to default when still empty. Server URL does not receive an implicit localhost default.

```
        if value := strings.TrimSpace(os.Getenv("ORCH_SERVER_URL"));
value != "" {
                cfg.ServerURL = value
        }
        if value := strings.TrimSpace(os.Getenv("ORCH_TOKEN"));
value != "" {
                cfg.Token = value
        }
        if value := strings.TrimSpace(os.Getenv("ORCH_NAMESPACE"));
value != "" {
                cfg.Namespace = value
        }
        if strings.TrimSpace(overrides.ServerURL) != "" {
                cfg.ServerURL =
strings.TrimSpace(overrides.ServerURL)
        }
        if strings.TrimSpace(overrides.Token) != "" {
                cfg.Token = strings.TrimSpace(overrides.Token)
        }
        if strings.TrimSpace(overrides.Namespace) != "" {
                cfg.Namespace =
strings.TrimSpace(overrides.Namespace)
        }
        if strings.TrimSpace(cfg.Namespace) == "" {
                cfg.Namespace = "default"
        }
        return cfg, nil
}
```

Exact source: internal/config/config.go:132–154.

For the Atlas experiment, imagine YAML naming `http://127.0.0.1:1` and namespace absent-config. Environment supplies another server value and absent-env. An explicit `--server` pointing at the owned server and `--namespace default` wins over both. Remove those flags and the

nonempty environment values win over YAML. Remove the environment settings and YAML wins. These are precedence rules for nonempty values, so an empty flag is not a reliable way to erase a configured credential.

Setting source	Priority	Atlas implication
Explicit nonempty flags	Highest	Select the intended server and namespace for this command
Nonempty ORCH_SERVER_URL / ORCH_NAMESPACE / ORCH_TOKEN	Middle	Override values in the config file
CLI YAML	Lower	Provides persistent operator settings
CLI fallback	Last	Namespace defaults; server remains required

The default configuration path comes from the operating system's user-config directory, under `orch/config.yaml`, with `.orch.yaml` as fallback if that directory cannot be determined. Automation should pass an explicit config path when it needs reproducible isolation. Do not repurpose HOME or dump the user's config file to prove which value was selected.

The implementation calls LoadCLI for server, token, and namespace separately. A malformed file can fail server selection before any request is made. That is different from successful configuration followed by 401. Report the failing boundary and preserve the command exit code, rather than telling the operator every problem is a bad token.

We tested server and namespace precedence with the actual CLI binary against the owned loopback server. Token precedence is source-inspected in the same loader; the lab does not claim an authenticated multi-token deployment because its fresh development server intentionally has user JWT enforcement disabled. Synthetic agent credentials used for reports are a separate internal contract and are never printed.

JSON Output for Operator Automation

Read the output writer before selecting fields. writeService emits the service object directly in JSON mode, and writeTasks emits the task array. These are not the HTTP envelopes containing service or tasks. An automation script that uses `.service.spec.replicas` against CLI inspect output is reading the wrong shape.

The status checker is `examples/teaching-extensions/ch08/atlas_status.py`. It runs `service inspect` and `service ps` with `--output json`, an explicit server, and selected namespace. It validates service identity, deployment version, replica count, unique task identities, task ownership, and required task fields. It counts only tasks with desired_status running and the current deployment version. Reported availability follows the upstream lifecycle helper: actual status running or healthy.

That choice deserves a label. Running means the server accepted a runtime execution report; it does not prove a business endpoint is responding now. Healthy records a health result at an observation time. Neither makes two separate inspect and ps requests a transactional snapshot. The script's JSON result explicitly says server observation only, no fresh host probe.

Its success condition is an active service whose number of current-version intended-running tasks equals spec.replicas, with all those tasks reported available. A zero-replica active service can satisfy that condition with zero current tasks. A task from another service is a contract error, while a historical task desired removed is excluded from the active count. This prevents scale-down history from being mistaken for a missing or extra current replica.

The exact child commands have the following structure, with the lab's ephemeral origin substituted by the harness:

```
orch --server "$ATLAS_SERVER" --namespace default --output json --
config "$ATLAS_CONFIG" service inspect atlas-api
orch --server "$ATLAS_SERVER" --namespace default --output json --
config "$ATLAS_CONFIG" service ps atlas-api
```

ATLAS_SERVER and ATLAS_CONFIG here are task-specific variables, not an implicit CLI default. The actual executed command arrays and the selected generated loopback address are retained in the lab record. The script never prints a bearer token or forwards raw subprocess stderr into its user-facing failure text.

There remains a failure window between its two reads. A concurrent rollout or scale can change the deployment version or replicas after inspect but before ps. The conservative outcome is a non-converged result and another bounded check, not automatically changing desired state. A robust deployment pipeline can poll with a deadline, but this teaching script intentionally performs one check and leaves retry policy to its caller.

Failures That Belong in Exit Status

The checker uses zero for reported convergence, one for a valid but non-converged observation, two for CLI/transport failure, and three for incompatible JSON shape. These are the teaching script's own statuses, not undocumented promises about every upstream CLI code. Upstream errors normally reach the executable's error path as a nonzero exit.

A two includes missing CLI, timeout, server rejection, and missing service resolution. Its concise instruction is to inspect CLI path, server reachability, namespace, and credential configuration. The reason for suppressing raw child stderr is specific: diagnostic strings can contain configuration details or URLs. Operators can run the corresponding CLI command in their own controlled terminal when they need the richer structured error.

A three means the script cannot safely interpret the current contract. It must not convert absent replicas or a renamed status field into zero and announce success. Likewise a malformed task row must not be silently skipped. Validating expected values after JSON decoding is the practical counterpart to chapter 7's renamed-field example.

The integration harness `examples/teaching-extensions/ch08/exercise_cli.py` verifies baseline hashes, starts one actual server on a newly selected loopback port, creates temporary config files, and executes the illustrated CLI/status procedures. Prerequisites are Python 3, the built pinned CLI/server, and local listener permission. No Docker daemon, database, or external endpoint is needed.

Run the actual lab procedure from the website root:

```
python3 books/inside-orch/examples/teaching-extensions/ch08/
exercise_cli.py \
  /tmp/inside-orch-labs-cbfc94ade5b5/bin/orch \
  /tmp/inside-orch-labs-cbfc94ade5b5/bin/orch-server \
  /tmp/inside-orch-labs-cbfc94ade5b5/orch
```

`validation/labs/ch08/exact-cli.json` records a passed execution. Its stdout retains every real child command, exit status, and diagnostic. It covers flag-over-environment-over-file, environment-over-file, file-only configuration, missing server failure, local validation, deployment, inspect, ps, missing service, and connection-refused status checks. Intended failures are assertions within a successful harness, not discarded errors.

Before any node exists, Atlas's three replicas are accepted as intent but the status checker exits one. The harness then registers a synthetic worker and authenticates running reports for its three assigned tasks. The checker exits zero on those server observations. A separate semantic fixture run in validation/labs/ch08/status-semantics.json rejects duplicate task identities, missing contract fields, foreign-service rows, and boolean replica values; it also checks zero replicas and old-version cleanup history. These controlled fixtures complement the real CLI exercise without claiming host observations. No Docker containers are

started, so the result cannot be described as three verified running host processes. This is an explicit and useful contract experiment rather than a simulated production deployment.

Cleanup stops only the process launched by the harness and removes its temporary config directory. The in-memory service disappears with that process. Captured evidence remains. The refused-connection case uses loopback port 1, not a remote URL or a previously running owner's endpoint. The chapter's commands deliberately avoid putting secrets on the command line.

The initial Cobra CLI commit established command structure. Configuration hardening later introduced shared config handling and related settings. Original prompts 05,43,44 explain the intended progression; the current parser, loader, command definitions, and actual child-command record establish the behavior we teach.

A shell caller should preserve the checker's status immediately instead of piping its output into an unrelated success-producing formatter. It can retain the JSON on standard output for a report while using the exit code to decide whether to proceed. A non-converged result should not trigger an unconditional redeploy: assigned replicas may simply be waiting for a worker, and a transport error leaves acceptance uncertain. Keep observation and mutation as separate operator actions. If a pipeline adds retries, give them a deadline and retain the last failure category so a missing server or incompatible schema cannot become an endless wait that looks like deployment progress.

Exercises

1. Predict the selected server and namespace when YAML, environment, and nonempty flags disagree. Explain the effect of removing each higher-priority source.
2. Explain why local validate succeeds without a server address but service ls fails. Identify which layer first needs an API client.

3. Modify your reasoning for an old-version healthy task alongside current-version assigned tasks. Should the status checker succeed, and which actual JSON fields decide?
4. An operator proposes grepping the table for running and treating any matching line as success. Explain the unsafe assumptions and how the exercised checker handles missing service, non-convergence, and transport errors.

See the worked answers.

Chapter 9 – Persistence, Transactions, and Optimistic Concurrency

Atlas API has a service ID, desired replica count and several task observations. Those facts currently live in a process. Adding a database beside that process does not answer the operational question: after the process dies, which component reconstructs the same authoritative state? Before promising recovery, we must separate a durable store implementation from an application that actually uses it.

This chapter assumes the executable trace from chapter 6 and the resource model from chapter 5. You will identify the row and transaction that protect an assignment, distinguish conflict from absence, and evaluate a database test without confusing its fixtures with server wiring. The concrete problem is two schedulers trying to assign one pending Atlas task. A second problem, deliberately kept separate, is restoring the service after a new server process starts.

The Store Contract and PostgreSQL Schema

The composite Store interface embeds node, service, task, deployment, event, secret, registry, namespace, quota and GitOps contracts. It is a useful inventory, but not the whole current domain: job, cron job, volume and notification interfaces are declared separately and are not embedded in that composite. This matters when imagining a durable replacement for MemoryService. Satisfying the older Store contract does not establish parity with every method the API currently calls. Claim ch09.contract.

Consider one task row from the initial schema. Its UUID primary key identifies the orchestration task; service_id references a service, and node_id can be null before placement. actual_status records an observation or assignment stage, while desired_status records intent.

The row has the desired service version, creation/update timestamps, restart information and a separate row_version. These identifiers serve different purposes. The service version describes application intent; row_version counts database mutation; neither should be casually relabeled a worker fencing epoch.

The original schema indexes tasks by service, node and status. Those indexes correspond to real lookup patterns: reconciliation asks for service tasks, the agent asks for node assignments, and the scheduler asks for pending work. An index makes a predicate efficient; it does not automatically make a multi-row decision serializable. Foreign keys and CHECK constraints provide another layer: they protect stored relationships and selected values even if a caller bypasses the Go API. They do not validate every lifecycle transition.

Later migrations change this initial model. Namespace uniqueness, image metadata and node agent_version are not present in the first migration. Reading 000001 alone therefore describes a historical starting point, not the complete baseline schema. The source lock contains 16 up/down pairs. When a test applies a hand-maintained subset, compare that list to fields selected by current SQL before treating a failure as an algorithmic bug.

Timestamps are normalized to UTC at the Go boundary, and SQL updates explicitly set an update time. TIMESTAMPTZ values describe instants rather than a display timezone. Preserve the value returned by the store as the next optimistic token; do not reconstruct it from a formatted dashboard string with lower precision. The current guarded writes compare expected_updated_at, even where an additional integer version is also incremented. A reader must inspect the predicate actually used, not infer it from a column name.

Transactional Boundaries and Conflict Errors

PostgresStore wraps a pool-backed postgresStore. Its WithTx method creates a transaction-scoped store whose db points at the transaction and whose begin field is unset. Operations invoked through that scoped

object participate in the same transaction instead of opening independent nested transactions. The callback's successful return permits Commit; an error returns without committing, and deferred rollback handles the unfinished transaction.

internal/store/postgres.go, lines 69–83 at the locked revision.

```
func (s *postgresStore) withTx(ctx context.Context, fn
func(*postgresStore) error) error {
        if s.begin == nil {
                return fn(s)
        }
        tx, err := s.begin(ctx)
        if err != nil {
                return mapPostgresError(err)
        }
        txStore := &postgresStore{db: tx}
        defer rollback(ctx, tx)
        if err := fn(txStore); err != nil {
                return err
        }
        return mapPostgresError(tx.Commit(ctx))
}
```

This is a meaningful boundary only if every relevant effect uses the transaction-scoped value. Calling the outer pool-backed store from inside the callback escapes that boundary. The generic store.WithTx helper also has a deliberate fallback: if its candidate does not implement Transactor, it simply calls the callback directly. That helps shared controller code work with fakes, but the function name alone cannot prove atomicity for every candidate. Claim ch09.transaction.

Suppose Atlas service creation and its domain event should be one database fact. If service insertion succeeds but event insertion fails, the transaction can roll both back when strict event emission propagates the error. Without that transaction, the service may exist without its corresponding history record. External Docker creation cannot join this SQL transaction; treating it as another callback statement would leave an unavoidable side-effect gap. The orchestration protocol needs reconciliation rather than pretending all systems share a commit.

The inspected transaction audit makes this distinction concrete. Commit 72fd9baff0bbfde49574257380bcd8f747cc2070 adds transaction-aware controller operations and strict events, including a rollback test. Earlier commit 7eba6544ffe2eba41b81faa811e899faf28b77a2 introduced the PostgreSQL layer. Their patches establish code evolution, not a production migration or successful execution of every original development prompt. Prompts 03 and 24 were added retrospectively; prompt 30 describes the later transaction audit.

Conflict handling is also part of the contract. A missing row, a stale optimistic token and an invalid state are not interchangeable. The scheduler treats a lost assignment race as a normal contention outcome and can continue; a transport or SQL error needs a different operational response. Retrying every error indefinitely would hide schema failures and could prevent useful work elsewhere.

Optimistic Writes Under Contention

AssignTask first obtains the current task according to its transaction context, checks whether pending→assigned is legal, serializes allocated ports, and issues a conditional update. The crucial predicate requires the same task ID, expected update time and actual_status pending. The update stores the chosen node and ports together with assigned state. Claim ch09.assignment.

internal/store/postgres.go, lines 912–927 at the locked revision.

```
        row := s.db.QueryRow(ctx, `
                UPDATE tasks
                SET node_id = $2,
                        actual_status = $3,
                        ports = $4,
                        updated_at = timezone('utc', now()),
                        row_version = row_version + 1
                WHERE id = $1 AND updated_at = $5 AND actual_status
= 'pending'
                RETURNING id, namespace, service_id, node_id,
container_id, desired_status, actual_status,
                        image, requested_image,
resolved_image_digest, image_registry, image_name, image_tag,
                        version, ports, conditions, restart_count,
failure_reason, created_at, updated_at, started_at, finished_at`,
                string(id),
                string(nodeID),
                string(types.TaskAssigned),
                portBytes,
                expectedUpdatedAt.UTC(),
```

Imagine schedulers A and B both read task T at time token U. Both compute a valid placement. A's update succeeds and changes T's status and update timestamp. B's later conditional update no longer matches. The acceptable result is one owner and one conflict, not two success responses with different node IDs. Replanning after the conflict is different from blindly repeating B's old write: the new snapshot may show that T is no longer pending.

This protects one task assignment. It does not, by itself, reserve all free CPU and host ports across a set of independently planned tasks. Two schedulers can each read available capacity, plan different pending tasks, and successfully claim both even if their combined plans oversubscribe the snapshot. Chapter 10 separates per-task uniqueness from global reservation. Increasing the number of predicates on one row cannot replace a deliberate cross-row allocation protocol.

Service updates follow a similar pattern: updateService writes the new spec and increments deployment_version only where the service ID, namespace and expected timestamp match. A concurrent operator holding an old token receives ErrConflict rather than silently replacing a newer specification. That is valuable lost-update protection, but the surrounding operation still needs to connect version history and events atomically. A successful SQL row update is not proof a worker is already running the new image.

Observation	What it establishes	What remains unproved
Conditional assignment rejects stale writer	One row did not accept both old decisions	Global CPU/port serialization
Service/event callback rolls back	Both SQL effects share the tested transaction	Docker-side rollback
Full schema migrations apply	Selected schema can be constructed	Default server uses that schema for live state
Controller restarts with one store object	Repeated controller logic can reuse retained state	Fresh-process recovery

Atlas Lab: Database Evidence Without a Silent Fixture Repair

Use only the disposable source copy and the dedicated labelled PostgreSQL container from the environment contract. The database name must end in _test because the upstream migrate helper refuses other names. The book's additional guard checks the loopback binding, exact database identity and ownership labels before permitting destructive tests. A suffix alone is not sufficient evidence that the database is disposable.

The actual guarded run is preserved in database evidence. WithTxRollsBackOnError passed. Integration and ConcurrentAssignTaskOnce failed with SQLSTATE 42703: current node queries require agent_version, while the reset helper applies only migrations 000001–000014 and the column arrives in 000016. This is an

executed fixture/schema mismatch. It is not a passing concurrency experiment, nor a reason to erase the failure and rerun altered upstream code under the same label.

```
python3 books/inside-orch/scripts/run_database_tests.py \
  --container <recorded-owned-postgres-container> \
  --working-copy /tmp/inside-orch-labs-cbfc94ade5b5/orch \
  --go /path/to/go1.25 --check-only
```

Run that from the website root; only remove --check-only after the scope check succeeds. Keep the designated URL in the environment rather than in copied commands or logs. The separate full-schema teaching fixture applied all 16 migrations and exercised AssignTask with two concurrent writers: one won and one received a conflict. It also preserved a coherent stored task. Its passing result is named separately from the failing untouched suite, and its fixture remains reviewable.

For a source-only alternative, inspect the conditional predicate and the rollback assertions, recording them as inspected rather than executed database outcomes. Clean up by stopping the exact owned container after all dependent experiments, then removing only its recorded dedicated volume if one was created. Never reset a shared developer database or use a broad volume-prune command.

Why the Default Server Is Still Not Durable

The executable still constructs NewMemoryService with envelope, policy and retention options. That object owns maps. Database configuration and schema checks do not redirect its methods into PostgresStore. The persisted tables can be correct and the default server can still lose its authoritative service state on process restart. This is the boundary that prevents a storage library from becoming an accidental operational promise. Claim ch09.composition.

A durable composition exercise therefore starts with a method inventory, not a constructor substitution. Identify service version history, node credentials, jobs, local volume attachments, notification sinks, operations and quota state. For each, choose the authoritative store and transaction

boundary. Then decide how controllers rebuild observations, how agents re-enroll without duplicating work, and how pending side effects are recovered. Persisting a service row while dropping its ownership or secret metadata can make restart more dangerous than an honestly disposable memory server.

The release decision should require a fresh-process test using an independently constructed application instance and the same durable database. Preserve container IDs and declared intent before termination, start a genuinely new process, and verify the restored API and reconciliation behavior. Do not reuse one MemoryService object and call the result database recovery. Until that composition exists and passes its own tests, describe PostgresStore as an implemented package and the running default service as memory-backed.

A useful contention record includes both input task snapshots and both outcomes, but excludes the database URL and authentication material. Compare the original UpdatedAt with the returned winning task, then re-read the stored row. The losing result alone is insufficient: an implementation that rejected both writers would also produce a conflict without making progress. The full-schema fixture checks that exactly one writer wins and the stored assignment is coherent. This provides positive and negative evidence together. It still says nothing about whether the chosen node has enough remaining capacity, because that is a different invariant with a larger read set.

Exercises

1. Trace one task row through AssignTask. Which predicate rejects a second stale writer, and why is row_version alone insufficient evidence of the guard?
2. Explain how a callback can accidentally escape WithTx even while executing inside its lexical body.
3. Classify the observed missing-agent_version failures. What separate fixture would isolate assignment correctness without rewriting the baseline?

4. Design the minimum evidence needed to claim durable Atlas service restart. Include one API state that is absent from the composite Store interface.

Worked answers. Exact source slices and their checksums are retained in chapter listings. The chapter's claims distinguish inspected implementation, the executed database fixture failure and the still separate durable-composition design.

Chapter 10 – A Deterministic Scheduler

Two Atlas API replicas need somewhere to run. A casual placement rule might select the first available node and hope Docker accepts the request. A deterministic scheduler instead makes its decision from an explicit snapshot, applies named constraints and returns a result that can be calculated and tested. Its value comes from knowing both what the algorithm guarantees and which live requests actually pass through it.

This chapter assumes resource quantities from chapter 5, executable composition from chapter 6 and guarded assignment from chapter 9. You will calculate a three-node placement, identify every eligibility filter and follow a plan into the store's conflict path. The key distinction is between a valid plan, a persisted task claim and enforced resource consumption. Those are three different pieces of evidence.

Candidate Ordering and Deterministic Choice

Plan accepts pending tasks, nodes, running tasks and a service map. It returns assignments without contacting Docker or writing to a database. This separation lets the same input be evaluated repeatedly, including with reordered input slices, while keeping the side effects outside the algorithm. The planner sorts tasks and nodes before iterating; it does not rely on map iteration or the incidental order returned by a store. Claim ch10.plan.

internal/scheduler/scheduler.go, lines 270–290 at the locked revision.

```
func Plan(input PlanInput) []Assignment {
        pending := sortedTasks(input.PendingTasks)
        nodes := sortedNodes(input.Nodes)
        state := newClusterState(nodes, input.RunningTasks,
input.Services)

        assignments := make([]Assignment, 0, len(pending))
        for _, task := range pending {
                if task.ActualStatus != types.TaskPending {
                        continue
                }
                service, ok := input.Services[task.ServiceID]
                if !ok && task.JobID != "" {
                        service, ok = jobTaskService(task)
                }
                if !ok {
                        continue
                }
                node, ports, ok := bestNode(task, service, nodes,
state)
                if !ok {
                        continue
                }
```

For each pending task, the planner finds its service specification. Finite job tasks can be represented through jobTaskService when they have a JobID and no ordinary service entry. Missing specifications cannot be safely guessed, so an unresolvable task is skipped. After choosing a node, the planner updates its temporary clusterState. Later tasks in the same invocation see the CPU, memory, ports and volume writers reserved by earlier decisions.

Candidate scoring is specific rather than a generic claim of balanced utilization. It first prefers more memory remaining after placement, then fewer tasks of the same service, then fewer total tasks, then lexical node ID. CPU participates in eligibility but is not the leading score. A node with slightly more remaining memory can therefore win despite a greater CPU load, provided the request still fits. Read the actual score comparator before calling the strategy round robin, bin packing or least loaded.

Stable tie-breaking has practical consequences. If two nodes have identical available memory and task counts, lexical ID decides. A shuffled store response should not change the result. Determinism makes a failure reproducible and a unit test precise; it does not guarantee that the snapshot remains valid until persistence finishes. Nor does it promise a globally optimal placement for all future workloads.

Ready Nodes, Labels, Capacity, and Ports

Eligibility starts with readiness. An offline or draining node is excluded even if it advertises abundant capacity. Volume mounts can require a particular node, and a writable ReadWriteOnce mount can have an existing owner. Placement labels are compared using the supported equality and inequality operators. An unknown operator is rejected rather than interpreted loosely.

Free resources equal advertised Allocatable minus resources accounted for by existing tasks. Requests use millicores for CPU and bytes for memory. They are not current utilization measurements. A task requesting 400m and 128Mi consumes those declared amounts in this accounting model whether its process is idle or busy. Container cgroup limits require a separate field path into Docker HostConfig, which the ordinary agent path does not complete at this baseline.

Port allocation adds another finite resource. A fixed published TCP port cannot be reused on the same node in one plan. Dynamic ports come from the planner's 30000–32767 interval, excluding reservations already seen in the snapshot or selected earlier in the current plan. A port failure can remain reserved in accounting to avoid immediately repeating an observed conflict. This logic concerns node-scoped host bindings; it is not a cluster-wide virtual service address.

The following fixture is calculated before running it. Atlas requests 400m CPU, 128Mi memory and fixed published port 31080. There are two pending tasks, atlas-01 and atlas-02, processed in ID order. Node b already runs an unrelated service using 100m, 64Mi and port 31080.

Node	Status	Allocatable CPU/ memory	Existing reservation	First Atlas result
a	offline	4000m /4096Mi	None	Rejected by status
b	ready	1000m /512Mi	100m /64Mi, TCP 31080	Rejected by fixed port
c	ready	1000m /256Mi	None	atlas-01 assigned

After atlas-01, node c has 600m and 128Mi free. atlas-02 would fit its CPU and memory exactly, but port 31080 is now reserved. Node b remains port-conflicted and node a remains offline, so the second task has no assignment. The limiting factor is the published port, not CPU. Changing the request's public port to dynamic would create a different experiment rather than retroactively changing this result.

The independent book fixture produced exactly that one assignment and confirmed that permuting node and pending-task input order left the plan unchanged. Executed hand calculation records the actual command, source pin and output. The fixture is a teaching addition compiled inside the disposable Orch module; it does not alter the scheduler or claim a real container was started.

One Assignment Under Concurrent Work

Scheduler.RunOnce reads pending tasks and ready nodes, loads existing task/resource state and services, then invokes Plan. It persists assignments individually. Each assignment carries ExpectedUpdatedAt from the task snapshot. persistAssignment passes that token to AssignTask and, where supported, combines the mutation with a strict assignment event in one transaction. Claim ch10.claim.

internal/scheduler/scheduler.go, lines 170–187 at the locked revision.

```
func (s *Scheduler) persistAssignment(ctx context.Context, task
types.Task, assignment Assignment) (Assignment, bool, error) {
        assignment.ExpectedUpdatedAt =
assignment.ExpectedUpdatedAt.UTC()
        if assignment.ExpectedUpdatedAt.IsZero() {
                assignment.ExpectedUpdatedAt = task.UpdatedAt.UTC()
        }
        transactional := false
        if _, ok := any(s.store).(store.Transactor); ok {
                transactional = true
        }
        assigned := false
        err := store.WithTx(ctx, s.store, func(txCtx
context.Context, tx Store) error {
                assignedTask, err := tx.AssignTask(txCtx,
assignment.TaskID, assignment.NodeID, assignment.Ports,
assignment.ExpectedUpdatedAt)
                if err != nil {
                        return err
                }
                assignedAt := assignedTask.UpdatedAt.UTC()
                if assignedAt.IsZero() {
                        assignedAt = s.now().UTC()
```

A conflict or a disappearing task is treated as a lost race rather than a fatal scheduler failure. The scheduler increments assignment-conflict metrics and omits that assignment from its successful result. Other store errors are returned with task context. This is useful differentiation: a competing worker's successful claim should not be reported as a database outage, while malformed SQL should not disappear as harmless contention.

The two-writer store experiment now has two distinct records. The untouched upstream suite failed before reaching assignment because its migration fixture omitted agent_version. The separate full-schema verification fixture applied all 16 migrations and observed one assignment winner and one conflict. Preserve both. The latter supports the tested conditional claim; it does not transform the former into a passing upstream suite.

Per-task uniqueness is narrower than globally safe reservation. Suppose two scheduler instances read the same free node capacity but plan different tasks. Each can win its own task's update. Unless capacity and port reservations are checked atomically across their shared state, the sum of both accepted plans can exceed the snapshot budget. The local clusterState protects ordering within one Plan call, while optimistic assignment protects ownership of one task. Neither scope should be silently expanded.

The failure window is between reading resources and persisting a set of choices. A third actor can create an assignment or bind a host port during that interval. A robust future design would need a shared reservation transaction, a serialized planner, or retryable revalidation of the relevant resource set. Such a change belongs in a separately tested teaching extension; it is not implied by the existence of an AssignTask transaction.

Planner Guarantees Versus Live Placement

The server does construct a scheduler loop, so it would be inaccurate to call the entire package disconnected. The important qualification is the normal MemoryService service path: reconcileServiceTasksLocked creates assigned tasks directly after choosing eligible ready/volume-compatible nodes. Those tasks are already assigned when the scheduler later asks for pending tasks. Claim ch10.active-path.

Consequently, the pure planner's CPU, memory and label filters do not automatically constrain every ordinary service deployment. A constrained comparison must send the same declared service/node intent through both paths and examine the resulting tasks. Creating a very large lab machine would hide the distinction; advertising a 500m/128Mi fake node and requesting 600m/256Mi makes it visible without stressing any real host.

Do not turn the comparison into a stronger accusation than the evidence supports. The planner can be internally correct while the application's ordinary service mutation path bypasses it. Conversely, showing one

bypass does not prove every task type avoids the scheduler: pending finite jobs and other package paths need their own trace. Classification is attached to the path and guarantee, not to an entire repository as a single badge.

The seeded property tests add breadth. They generate varied node readiness, labels, resource requests and ports, then assert repeatability, unique task assignment and capacity/port invariants. Fixed seeds make a failing generated case reproducible. Those assertions remain about Plan's supplied model; they do not contact the Engine, validate kernel throttling or model every race between separate planner instances.

Development History and Atlas Experiment

The scheduler began with fce0a657651c00ffb697bf4a4aba12d0b171f463. The concurrency audit 1a1c216e8cfb442465f4d3c2c114d20da0342795 introduced guarded claim handling and related metrics; 3bfcef82a54dec457cf9d4187705fa70cc7215cf added port-awareness changes. Their actual patches, not their titles alone, connect these concepts to implementation. Original prompts 09 and 17 are retrospective development intent; prompt 31 is the later concurrency request.

To reproduce the hand calculation, copy the canonical teaching fixture under examples/teaching-extensions/ch10 into book-scheduler beneath the disposable source module. Use the pinned Go toolchain and run-owned caches. No Docker, database, operator token or service process is required.

```
GOTOOLCHAIN=local GOFLAGS=-buildvcs=false go run ./book-scheduler
```

Run that only from the disposable Orch copy, after copying the fixture and verifying the source bytes. Expected success is one assignment for atlas-01 to node c, atlas-02 absent, and an unchanged result after input permutation. The actual recorded run passed. Remove only the book-scheduler teaching directory when finished; never modify or build in the immutable snapshot.

For further exploration, change a copy of the fixture to dynamic ports and predict both assignments before executing it. Record that as a new fixture result. A live Docker comparison is optional and must use the isolated daemon contract because workload published ports can expose all interfaces. Do not remove a host container merely because it occupies the port needed by your experiment.

A failed placement should preserve enough input to explain its rejection. For Atlas, an operator needs node readiness, advertised capacity, existing reservations, the requested port and the task's expected update time. A bare unable-to-place event cannot distinguish a port collision from insufficient memory. Keep a sanitized snapshot beside the planner output when investigating the lab. Avoid substituting live CPU percentage for declared requests: doing so changes the model being evaluated.

Now remove only the existing reservation on node b in a copied fixture. Both ready nodes become eligible for the first task, but node b has more memory remaining after placement and wins that comparison. Its chosen fixed port then prevents the second Atlas task from sharing node b, while node c remains available. Predict two assignments to different nodes before running the changed fixture. This small change demonstrates why a constraint must be applied before a score: a high-scoring node with a conflicting fixed port was previously ineligible, whereas the same node without that reservation can participate. Keep the modified result separate from the recorded baseline experiment.

Exercises

1. Calculate why node b loses despite having more free memory than node c. Which condition precedes scoring?
2. Change only Atlas to two dynamic published ports. Predict whether the second task can now fit node c and which score comparison matters.
3. Explain how two successful claims for different task IDs can still violate a shared capacity assumption.

4. An operator cites the seeded property suite to claim every deployed Atlas container is CPU-limited. Identify the two missing evidence edges.

Worked answers justify each result. The operational decision is to name planner invariants precisely, retain the per-task conflict guard, and require an explicit integration/reservation trace before promising placement or runtime enforcement beyond this tested scope.

Chapter 11 – Reconciliation as Repeated Convergence

Atlas API should have two replicas, but the store currently contains one running task and one failed task. The operator's intent has not changed. Something must decide whether to create another task, preserve the failure, or wait for an in-progress operation. Reconciliation is that repeated comparison between intent and recorded state. Its correctness depends on counting the right objects and assigning each side effect to the right owner.

You already know the task lifecycle, MemoryService composition and optimistic store boundary. Here you will predict what successive iterations read and write, explain why pending tasks count before Docker starts, and distinguish controller restart from process recovery. The standalone reconciler package is an implemented library; the default server's immediate MemoryService path must be traced separately. Claim ch11.composition.

Replica Convergence Without Docker Calls

The reconciler's Store interface exposes services, task collections, active deployments, task creation/stopping, service status changes and events. It contains no Docker Runtime. That boundary is deliberate and visible: the controller decides desired work in the store, the scheduler chooses placement for pending work, and the agent acts on containers. Confusing these roles would make a successful task-row insertion sound like a running application.

ReconcileOnce first checks cancellation and store presence, then acquires its configured LeaderLock and releases the lease when the iteration returns. The default lock is NoopLeaderLock. An interface named LeaderLock does not demonstrate distributed exclusivity, and a fake lease test only proves that the expected call was made. Actual shared leadership, session cleanup and fencing are considered in chapter 34.

The internal reconcile pass loads services and a set of services with pending, running or rolling-back deployments. It sorts services by ID and dispatches based on service status. Active services with an active rollout are skipped, preventing the ordinary replica loop from fighting the rollout controller. Deleting services follow a different path; deleted services are not treated as active intent. A final orphan scan stops nonterminal tasks whose service is no longer known.

For an active service, reconcileService loads and sorts its tasks. Current-version nonterminal tasks enter the current set; nonterminal tasks from another version enter the outdated set and are stopped. Failed current-version tasks with a non-restartable policy are counted separately. This last detail prevents a never-restart workload from being recreated endlessly just to satisfy a replica number. Claim ch11.counting.

internal/reconciler/reconciler.go, lines 304–322 at the locked revision.

```
        for _, task := range tasks {
                if task.Version != service.DeploymentVersion {
                        if isNonTerminal(task) {
                                outdated = append(outdated, task)
                        }
                        continue
                }
                if isNonTerminal(task) {
                        current = append(current, task)
                        continue
                }
                if task.ActualStatus == types.TaskFailed && !
restartAllowed(service.Spec.RestartPolicy) {
                        nonRestartableFailed++
                }
        }

        for _, task := range stopOrder(outdated) {
                if err := ctx.Err(); err != nil {
                        return result, err
```

The effective count is current nonterminal work plus non-restartable failures. If that count is below desired replicas, the controller creates the deficit as pending tasks. If the current set exceeds desired replicas, it

selects extras to stop. Pending, assigned, pulling and created tasks are not absent replicas merely because they are not yet healthy. Counting only running tasks would cause every slow image pull to produce more tasks on the next iteration.

Notice the difference between replica convergence and availability. Two pending tasks can satisfy the controller's work-count requirement while zero tasks serve HTTP. Health and rollout accounting use additional observations for availability decisions. A dashboard that collapses these dimensions into one green replica count would hide a real operational distinction.

Repeated Iterations and Restart Idempotency

Consider a fresh fixture with Atlas desired replicas two and no tasks. The first pass reads an empty task list, calculates a deficit of two and creates two pending tasks. The second pass reads those pending tasks, counts both as current work and creates nothing. Its successful outcome is a no-op, not another pair of creation events. Claim ch11.repeated.

Now replace the Reconciler object between passes while keeping the same fake store. The new controller still sees the pending tasks, so it also creates nothing. The upstream restart-hardening test does exactly this: New(store), one iteration, another New(store), another iteration. That is useful controller reconstruction evidence. It is not a fresh process, a database reload or a persisted-state recovery test.

The distinction explains what the restart invariant actually requires. The controller must derive its next action from stored tasks rather than an unpersisted local list of tasks it remembers creating. If an action's result is already recorded, a restarted controller should recognize it. If the store itself disappears, there is no retained result to recognize. Default MemoryService process loss therefore remains outside the proof.

Creation and its event can share a transaction when the supplied store supports Transactor. Without it, the generic helper executes directly and event emission is best effort. There is also no automatic transaction

covering the entire read-count-create sequence across all services and controller instances. Two independent controllers can read the same deficit before either writes. A local idempotence test is not a substitute for durable ownership or a cross-controller serialization strategy.

This is the concrete failure window: pass A and pass B both observe one missing replica; each creates a distinct pending task. Task IDs are unique, but the total desired work is now excessive. A later iteration may scale down, yet temporary over-creation still happened. The appropriate claim is eventual correction under the tested model, not an invented exactly-once creation guarantee under arbitrary concurrent controllers.

Failed Tasks, Restart Policy, and Scale-Down

For Atlas with restart on failure, one running task and one failed task against desired count two yields one replacement. The following pass includes that replacement's pending state and does not create another. The failed historical task remains useful evidence, but it no longer inflates the active count. The executed tests verify both the single replacement and the absence of duplicate replacement events.

With restart policy never, the failed task instead contributes to effectiveCount. That choice respects the configured no-restart intent even though it leaves the service below running availability. The algorithm's restartAllowed helper accepts empty, always and on_failure conditions and rejects others. Do not invent an exponential delay, retry budget or backoff schedule merely because the domain RestartPolicy type has additional fields; trace what this particular controller reads.

Scale-down similarly needs stable selection. stopOrder begins with stable task sorting and applies stop priorities, using descending task ID as a tie-breaker. The fixture with three running tasks and desired one stops task-c then task-b. The next pass does not issue the same desired-stop mutations again. This makes tests reproducible and avoids gratuitously changing which replica is retained.

internal/reconciler/reconciler.go, lines 435–446 at the locked revision.

```
func (r *Reconciler) stopTask(ctx context.Context, task types.Task,
reason string) (bool, error) {
        if task.DesiredStatus == types.TaskStopped ||
task.DesiredStatus == types.TaskRemoved {
                return false, nil
        }
        transactional := r.transactional()
        stopped := false
        err := store.WithTx(ctx, r.store, func(txCtx
context.Context, tx Store) error {
                stoppedTask, err := tx.StopTask(txCtx, task.ID,
task.UpdatedAt)
                if err != nil {
                        return err
                }
                if err := events.Emit(txCtx, tx, types.Event{
```

stopTask first returns a no-op if desired state is already stopped or removed. Otherwise it attempts a guarded store update using the task's UpdatedAt. A lost conflict or missing task is treated as no completed stop, while other failures are returned. The controller reports only successful desired-state changes in its counters. It has still not issued Docker StopContainer; the agent owns that later action.

Deletion extends the same separation. A deleting service remains deleting until its tasks report actual removed state. If the agent is absent, changing intent does not physically remove a container. A safe interface should show the intermediate state rather than claiming deletion completed merely because the HTTP request was accepted. The chapter 18 cleanup trace follows that sequence in detail.

How This Controller Is Invoked

The default server constructs MemoryService and calls immediate service reconciliation as part of its own mutations and observations. It does not construct internal/reconciler.Reconciler. Inside reconcileServiceTasksLocked, ordinary missing replicas are created

directly with a node and assigned state, using ready/volume-compatible nodes. That differs from this library's creation of pending tasks for a separate scheduler.

Both paths address desired-state convergence, but their placement, restart and transaction contexts are not interchangeable. A reader who searches only for the package named reconciler can miss the actual executable behavior. Conversely, a reader studying only the memory path would miss a useful independently testable library. Keep the call graph attached to each explanation instead of declaring one conceptual diagram true for all paths.

Question	Standalone reconciler	Normal MemoryService path
Where is replica work decided?	ReconcileOnce over Store snapshots	reconcileServiceTasksLocked under memory mutex
How are missing tasks introduced?	Pending desired-running tasks	Directly assigned tasks when eligible nodes exist
Does this code call Docker?	No	No; agent still owns runtime actions
What survives a default process restart?	Depends on supplied store	Process-local maps do not survive

The history shows why these distinctions matter. 208eb99758a5dac15a3e083fce3fc3e0dcfdd2e3 introduced the reconciler implementation. a11cec091a69720c14b9713d8be68f38fe6df332 added active-deployment skipping and restart/no-op regression coverage. 316ba80ac1ea7ba465f1b0826b655233a651c043 hardened task-state idempotency in the control plane. These are related improvements, not proof every package shares one invocation path. Original prompts 10 and 24 are retrospective intent; prompt 32 names the later restart audit.

Atlas Lab: Observe Reads Before Counting Success

The source-only experiment needs Go 1.25 and the disposable pinned source copy. It does not need Docker, a database or an active server. Run the focused upstream fixtures from that copy, with external integration flags unset and the selected GOROOT/cache configuration from the environment contract.

```
go test ./internal/reconciler -run TestReconcile -count=1 -v
```

The actual focused run passed. Its raw output names the repeated no-op, controller reconstruction, failed-task replacement and scale-down cases. The observation is constrained to fake-store controller behavior; it is not an Atlas container deployment or database process restart.

When reading each test, write three columns: tasks read on pass one, mutations emitted on pass one, and tasks read on pass two. For the two-missing-replica case, the last column must include the pending tasks created earlier. For the failed-replacement case, it must include the new task as current work. This exercise detects whether the test actually models retained state rather than merely counting calls on a mock that returns the same old snapshot forever.

A further safe extension can place a barrier between two reads to expose simultaneous deficit decisions. That experiment should assert the actual over-creation or conflict behavior, not weaken its expectation to obtain a green test. Keep its source under teaching-extensions and record which store/lock it supplies. Cleanup consists only of test process termination and optional removal of copied teaching files; no host containers are created.

A reconciliation result is also a statement about the state that was read. If Atlas changes replicas while a pass is computing its deficit, a later pass must observe that newer intent. The first pass should not be presented as a snapshot of the cluster at the instant its result is displayed. In a safe race fixture, pause after ListTasksByService, modify the desired service in

the fake store, then release the pass and inspect both the emitted mutations and the next pass. Such a test can reveal stale intent without creating containers. Record which object revisions the fixture preserves; simply asserting eventual replica count can hide an unnecessary intermediate creation or stop.

Exercises

1. Atlas has desired two, one running task and one pulling task of the current version. Predict the next create count and explain why health is irrelevant to that count.
2. Repeat the calculation with one running and one failed task under restart never, then under on_failure.
3. Identify what survives in the controller-restart fixture. Why would deleting its fake store invalidate the claimed guarantee?
4. Design a two-controller race fixture and state which additional mechanism would be required before calling creation globally unique.

Worked answers include the counted sets and the concurrency reasoning. Use the library to study convergence, keep observed state separate from actual container availability, and require the executable call path before transferring its invariants to a live deployment.

Chapter 12 – Idempotent Docker Operations and Drift

An Atlas task was created, but the agent lost the response before the control plane recorded its container ID. The next polling pass asks for the same work again. If create always means allocate another container, retries become a duplication mechanism. If create instead searches for the task's existing runtime identity, a retry can converge. The difference lives in concrete labels, lookup behavior and error handling rather than in the word idempotent.

You should already know lifecycle ownership and the executable's memory boundary. This chapter follows the Runtime interface into the Engine adapter and back through agent adoption. You will predict repeated operations, explain a manually stopped or missing container, and reject cleanup that cannot establish ownership. Docker socket permission is part of the experiment's privilege boundary, not a harmless prerequisite to tick off.

A Typed Runtime Boundary

The runtime interface names image pull, container create/start/stop/remove, inspection, managed-container listing and log streaming. Health probes belong to the agent's separate checker dependency, rather than a Runtime interface method. Its arguments use typed ContainerSpec, ContainerID and status structures. Controllers can express intent without importing Engine client details, while the agent owns the adapter that actually changes containers. Claim ch12.runtime.

ContainerSpec contains more than image and name. It can carry environment, published ports, labels, resource limits, security context, command and volumes. Support in that type is not proof that every application path populates every field. The agent copies task.Command

and volume mounts into the spec, but omits Resources in its creation literal at this baseline. Direct adapter resource tests and ordinary service enforcement are therefore distinct claims.

The Engine adapter validates the spec before asking Docker to create it. It builds configuration and HostConfig using the supplied fields, then passes them to the client. An invalid image or identity should fail before an ambiguous side effect. A successful create response yields a container ID, but running is a separate operation; the agent reports created before StartContainer and running afterwards.

FakeRuntime implements the same interface with in-memory containers, an operation log and injected failures. Its create path is synchronized and keyed by task ID. Tests can assert a sequence such as pull, create, start without needing a daemon. The fake does not model kernel networking, cgroup enforcement, registry availability or every Engine error. Passing a fake test is valuable protocol evidence within that model, not a substitute for real adapter integration.

Managed Labels as Resource Identity

managedLabels copies supplied labels and overwrites the orchestrator's identity labels: managed marker, service ID, task ID, node ID and version. These labels allow the agent and adapter to rediscover a container when an earlier response was lost. Container name alone is not the identity contract; names can collide and a manually created container can resemble an expected name.

internal/docker/docker.go, lines 486–497 at the locked revision.

```
func managedLabels(spec ContainerSpec) map[string]string {
        labels := make(map[string]string, len(spec.Labels)+5)
        for key, value := range spec.Labels {
                labels[key] = value
        }
        labels[ManagedLabel] = "true"
        labels[ServiceIDLabel] = spec.ServiceID
        labels[TaskIDLabel] = spec.TaskID
        labels[NodeIDLabel] = spec.NodeID
        labels[VersionLabel] = strconv.FormatInt(spec.Version, 10)
        return labels
}
```

The Engine create operation first searches for a managed container with the same task ID. If it finds one, it returns that ID rather than creating again. If ContainerCreate returns a conflict, it repeats the lookup to recover a possible concurrent creation. Other lookup or create errors are propagated. Claim ch12.create.

That lookup is narrower than a complete spec comparison. The adapter's task-based idempotence must be combined with the agent's richer adoption checks before treating a discovered container as correct current work. The agent compares managed/node/task/service/version metadata and, when available, image identity. It does not compare every environment value, port binding, mount or security field. Exact task identity and full desired-spec equivalence are different invariants.

The ordinary service case uses a nonempty ServiceID. Finite jobs introduce an additional edge: taskWorkloadID labels their workload as job-<JobID>, while the shared matching helper compares the service label with task.ServiceID. That discrepancy needs its own job adoption test; do not extend a service-adoption result to every workload merely because both use the same create function. Chapter 31 follows the finite-work path separately.

An untrusted actor with Docker socket access can manipulate labels and runtime objects. Labels are useful engineering metadata, not a security boundary against that actor. Treat the worker and its daemon as part of

the trusted computing base. A cleanup command must not assume that every container whose label says orch.managed belongs to this particular book run.

Repeated Create, Start, Stop, and Remove

CreateContainer's first retry can return the existing ID. StartContainer treats Docker's not-modified response as success, which lets a repeated start converge on already-running state. StopContainer first inspects; a missing or already-stopped container is successful completion of the desired stop. RemoveContainer similarly treats not-found as success. Claim ch12.operations.

internal/docker/docker.go, lines 191–219 at the locked revision.

```
func (r *EngineRuntime) CreateContainer(ctx context.Context, spec
ContainerSpec) (ContainerID, error) {
        if err := spec.Validate(); err != nil {
                return "", err
        }

        existing, err := r.findContainerForTask(ctx, spec.TaskID)
        if err == nil {
                return existing.ID, nil
        }
        if err != nil && !errdefs.IsNotFound(err) {
                return "", err
        }

        config, hostConfig, err := dockerContainerConfig(spec)
        if err != nil {
                return "", err
        }
        response, err := r.client.ContainerCreate(ctx, config,
hostConfig, &network.NetworkingConfig{}, nil, spec.Name)
        if err != nil {
                if errdefs.IsConflict(err) {
                        existing, findErr :=
r.findContainerForTask(ctx, spec.TaskID)
                        if findErr == nil {
                                return existing.ID, nil
                        }
                }
                return "", fmt.Errorf("create container for task %s:
%w", spec.TaskID, err)
        }
        return ContainerID(response.ID), nil
}
```

These semantics eliminate several normal retry hazards. They do not mean every call is free of side effects or that any error can be ignored. A permission failure, unavailable daemon or incompatible configuration still matters. In particular, CreateContainer must not interpret an arbitrary inspection failure as proof the container is absent. The agent fixture verifies that Docker unavailability does not trigger reckless recreation.

There is also a separation between repeating the same operation and changing intent. Reusing a task ID with a different version or image is not necessarily an idempotent retry. The agent can identify mismatched managed metadata, remove a container that belongs to the same assigned task/node scope, and then create correct work. Such replacement is an explicit convergence decision, not a hidden property of create.

The creation/reporting failure window is easy to describe. Docker creates container C; the agent tries to report created with C's ID; the HTTP request fails or its response is lost. On a later pass the server can still present a task without that ID. The agent must search labels and adopt C before creating another. A retained task ID makes that recovery possible; a fresh default server that forgets the task entirely is a different process-durability problem.

Repeated request	Baseline adapter response	Important limit
Create same task identity	Return existing managed container when found	Lookup is not full spec equivalence
Start running container	Not-modified accepted	Other Engine errors returned
Stop stopped/ missing container	Success	Ownership must already be established
Remove missing container	Success	Remove can delete attached anonymous volumes

The remove implementation requests RemoveVolumes. That makes cleanup scope particularly important for storage experiments. A teaching fixture should never attach a user's real data merely to demonstrate retry behavior. Keep local volume cases explicitly separate, with exact run-owned names and a clear record of which data is disposable.

Drift From Manual Docker Changes

A user can stop or delete a managed container outside the agent. currentTaskContainer first inspects the task's recorded container ID, then falls back to managed node/task lookup. A missing container generates a drift observation when task state implies one should exist. A matching stopped service container can be started again; a nonzero exit is instead reported failed according to the execution path.

The agent records three drift categories: managed container missing, unexpected managed container and managed-container state mismatch. Those categories describe observed discrepancies. They do not imply a centralized durable incident store; the configured recorder/logger determines where the observation goes. A test can verify the category without claiming an operator actually saw a production alert.

Removal requires its own identity check. A recorded ContainerID can point at a nonmanaged or differently owned object; the agent must not blindly remove it because the task says so. removableAssignedContainer checks the managed marker plus node and task ownership. The focused fixture verifies that a nonmanaged recorded container is not deleted. Claim ch12.cleanup.

cleanupUnassigned lists managed containers for the current node and removes those whose task is absent from the fetched assignment set. This is meaningful only if that assignment set is successfully obtained and authoritative for the scope. In the inspected code, a failure to list assignments returns before cleanup. A temporarily unreachable server therefore differs from a successful empty assignment response. That distinction protects against one dangerous outage-to-deletion conversion.

A successful empty set after a fresh memory-backed server restart remains a different risk: the server may have genuinely forgotten tasks while containers still exist. The agent cannot infer the former desired state from the missing API records alone. The safe lab records exact

container IDs before restart and separates this scenario from ordinary drift repair. Do not promise that container adoption repairs lost authoritative service intent.

History and Atlas Lab

The Docker adapter entered the project in 2969dfcebd0b15dea99cdf97b5ccdd1ddfd36b41. The deterministic fake arrived in c69f9ffa830284692aabf9e65ab9eb1f79d3a114, and cf9769553d6599d798fafaffe1495d3d739fe993 added drift-specific recording and stronger matching. Original prompt 06 is retrospective intent; prompts 33 and 41 relate to the later fake and drift work. The patches show changed code and assertions, not an automatic record of a live Docker experiment.

The safe source-only lab runs the adapter-client and FakeRuntime tests together with the agent's missing/stopped/mismatched/nonmanaged cases. Use the disposable source copy with Go 1.25 and no external integration flags. The focused execution record passed and names its exact selection. No daemon or host containers were used by this run.

```
go test ./internal/docker ./internal/agent \
  -run 'TestFakeRuntime|TestCreateContainer|TestStopContainer|
TestRemoveContainer|TestReconcileAssignedTask(Recreates|Restarts|
Replaces)|TestReconcileIgnoresNonManaged|
TestReconcileDoesNotRemoveNonManaged' \
  -count=1 -v
```

Read the resulting operation strings alongside created-spec and container-count assertions. Two create requests can appear in an operation log while only one container exists; counting calls alone would misclassify idempotence. For a manually missing container, expect a new container for the same orchestration task and an updated report. For an unrelated nonmanaged container, expect no removal operation against its ID.

An optional real experiment needs a dedicated disposable daemon or VM and the environment contract's exact ownership record. Stop only the recorded Atlas container, observe the agent's response, then delete only that same owned ID to exercise missing-state repair. Keep image ID, task ID, labels and before/after inspection in the log. Stop the agent before final cleanup so it does not recreate what the cleanup removes. Never use docker system prune or a global managed-label deletion.

Consider a manual label change on an otherwise running Atlas container. A task-ID lookup can now fail to find the object, while its process still owns a published port. Creating another container may then produce a port-binding failure rather than restoring service. The right incident record includes the task's recorded container ID, the inspected object's ownership labels and the exact Engine error. A broad search by image name would be dangerous because another workload can use the same image legitimately. Ownership mismatch is therefore both a convergence problem and an authorization boundary.

For the lab, use a fake runtime to present that mismatch first. Assert which object was inspected and that the unrelated object was not stopped or removed. If the same scenario is later repeated on a real daemon, create the altered object within the run's resource inventory and restore or remove only that object. A successful replacement does not justify sweeping all containers with similar names, nor does a failed replacement justify changing the ownership predicate to accept arbitrary running processes.

Exercises

1. Why can two CreateContainer operation records still establish an idempotent result? Name the additional assertion needed.
2. Describe the safe response when InspectContainer returns a daemon transport error rather than not-found.
3. A task's ContainerID points at a user's nonmanaged container. Trace the checks that prevent deletion and explain why the task record alone is insufficient.

4. Design a real adoption experiment after a lost created-report response. State what it proves and why it does not prove default-server process durability.

Worked answers distinguish task identity, runtime identity and desired-spec comparison. Preserve those distinctions when interpreting drift, and keep the authority to touch Docker resources narrower than the convenience of a broad label selector.

Chapter 13 – Agents, Heartbeats, and Task Reports

The server has decided that Atlas task T belongs on node N. Only the worker can observe whether Docker actually created its container. That split turns a simple deployment into a protocol: enroll the worker, maintain its identity, fetch assignments, apply local effects and report observations. Every network response can be delayed or lost, so the protocol must be read as a sequence of potentially ambiguous steps rather than one successful RPC.

This chapter assumes lifecycle guards, runtime adoption and the default server's state boundary. You will trace one enrollment and assignment, distinguish the agent's local authority from the server's desired-state decisions, and reason about a lost running report. The goal is a concrete delivery model with explicit trust and cancellation limits, not an invented exactly-once guarantee.

Enrollment and Agent Credentials

The agent executable loads configuration, constructs its runner and starts the logging/runtime side of the process. Runner.Run validates configuration and creates an HTTP client or Engine runtime when they were not injected. It detects capacity, registers with the server and keeps the returned node identity for later heartbeat and task operations. Claim ch13.enrollment.

The bootstrap token authorizes initial registration. The server can issue an agent credential whose plaintext is returned to the worker while a hash, expiry and revoked flag are stored with node identity. The token issuer generates 32 random bytes and encodes them; its default lifetime is 15 minutes. The credential validator checks revocation, token/hash equality and expiry against the supplied current instant. It does not turn an HTTP connection into mutual TLS.

internal/auth/agent.go, lines 44–57 at the locked revision.

```
        raw := make([]byte, 32)
        if _, err := rand.Read(raw); err != nil {
                return AgentCredential{}, AgentCredentialRecord{},
fmt.Errorf("generate agent credential: %w", err)
        }
        token := base64.RawURLEncoding.EncodeToString(raw)
        expiresAt := time.Now().UTC().Add(i.ttl())
        return AgentCredential{Token: token, ExpiresAt: expiresAt},
AgentCredentialRecord{
                Hash:      HashAgentToken(token),
                ExpiresAt: expiresAt,
        }, nil
}

func (i *TokenAgentCredentialIssuer) Validate(_ context.Context,
token string, record AgentCredentialRecord, now time.Time) error {
        if record.Revoked {
```

The HTTP client stores a credential returned by registration or heartbeat and reuses it on subsequent requests. Bootstrap and enrolled credential roles are therefore distinct. A copied development token in every operator example would hide that distinction and unnecessarily spread credentials into shell history. Lab commands keep disposable secrets in the process environment and execution evidence records only environment key names.

Identity is also bounded by state ownership. With the default MemoryService, the node's credential record lives in the server process. Restarting that process can invalidate the worker's relationship even if Docker containers continue running. Do not describe credential reuse as durable enrollment across arbitrary server loss; inspect the re-registration and rejection behavior separately.

A worker holding Docker access belongs to the trusted computing base. Even strong API authorization cannot prevent a compromised socket-capable agent from reading or altering its own host's workloads through Docker. The protocol authenticates supported requests; it does not

create a kernel or daemon privilege boundary between the agent and the host. Production trust choices require more than changing a bootstrap secret.

Heartbeat, Polling, and Assigned Tasks

Runner.Run uses a timer derived from HeartbeatInterval with jitter. On each timer event, it performs a heartbeat with retry, updates node state from the response and then reconciles assigned tasks. Assignment polling is therefore sequenced behind the heartbeat. A heartbeat operation that keeps failing can delay task reconciliation; the source does not start an independent perfectly reliable task stream alongside it.

The heartbeat advertises detected CPU/memory as both capacity and allocatable. CPU derives from runtime.NumCPU; memory reads MemTotal on Linux. This is host detection, not an administratively reserved budget after subtracting every unrelated process. The scheduler's declared accounting and the agent's capacity advertisement should not be confused with live cgroup utilization.

Retry behavior is specific. registerWithRetry and heartbeatWithRetry wrap operations in the retry helper. It starts from 250 milliseconds plus randomized delay, doubles backoff while below its threshold and exits when the context is canceled. It does not classify permanent authentication errors separately from transient network errors. A token rejection can therefore occupy the retry loop instead of immediately triggering a different recovery path. Claim ch13.retry.

internal/agent/agent.go, lines 792–812 at the locked revision.

```
        for {
                err := operation()
                if err == nil {
                        return nil
                }
                delay := backoff +
time.Duration(rng.Int63n(int64(backoff)))
                timer := time.NewTimer(delay)
                select {
                case <-ctx.Done():
                        timer.Stop()
                        return ctx.Err()
                case <-timer.C:
                }
                if backoff < 5*time.Second {
                        backoff *= 2
                }
        }
}

func DetectCapacity() types.Resources {
        return types.Resources{
```

After a successful heartbeat, reconcileAssignedTasks obtains the assignment collection. If listing fails, it returns before cleanup. On success it builds a map of assigned task identities, attempts each task and finally cleans up managed containers absent from that map. This distinction matters: no successful response is not the same as a successful response containing no tasks. Only the latter supplies the set used for cleanup.

The server owns assignment, desired stop/remove and lifecycle acceptance. The agent owns inspection and local Runtime calls. A task's reported running state is an observation about one container, not permission for the worker to choose arbitrary service intent. NodeID and task ownership checks at the API/control-plane boundary prevent one enrolled node from reporting another node's work as if it were its own.

Local Container Convergence and Reports

For desired stopped or removed work, ensureTask takes the removal path. For desired running work, it first tries to rediscover a matching container. If a matching container is running, it can re-report running when the task record lacks the correct container ID or remains in an earlier transition state. This is the recovery path for an agent restart that loses local memory but retains server task identity and labelled runtime objects.

When no matching container exists, the agent reports pulling, pulls the selected image with supplied registry auth, creates a typed ContainerSpec, reports created, starts the container and reports running. Health observations happen after those execution steps. The request carries NodeID, status, ContainerID where known, failure reason where applicable and exit code for finite completion/failure. Claim ch13.reports.

Those reports are individual requests. reportTaskStatus itself does not call the generic retry helper; it sends once and increments a state-change metric on success. Recovery can occur through a later reconciliation pass, which re-observes the container and task state. Saying every status request is automatically retried with the registration backoff would misdescribe the implementation.

A failure after Docker create but before a successful created report leaves an ambiguity: the server may not know the container ID, while Docker already has the side effect. The label-based lookup from chapter 12 lets the next pass recover it. A failure after the server accepted running but before the client received the response can lead to another running report. Lifecycle guards and idempotent terminal duplicates determine which repeats are allowed.

Boundary	Writer/observer	Relevant evidence
Desired service replicas	Operator/server	Service mutation and reconciliation
Node readiness	Heartbeat/server monitor	LastHeartbeatAt and timeout handling

Boundary	Writer/observer	Relevant evidence
Actual container state	Agent/Docker	Inspect and status request
Acceptance of task report	Server	Ownership and lifecycle guards
Physical stop of an isolated old worker	Not established by missing heartbeat	Requires fencing beyond replacement

The failure reason is also part of the trust boundary. Runtime errors can contain operational details, and generic log statements can spread those details if callers include credentials. The existence of redacted configuration views does not sanitize every arbitrary error string. Keep registry auth and bearer values out of experiment output, and inspect each report/log field rather than claiming universal secret redaction.

Dropped Reports and Cancellation

Consider a concrete timeline. The agent starts container C for task T and sends running. The server accepts it, but the response disappears. The agent returns an error to the local reconciliation loop and logs a warning. On the next successful heartbeat/poll, it inspects C, compares task metadata and may recover without creating another container. This is repeated convergence under ambiguous acknowledgement, not exactly-once delivery.

Now insert an operator delete before that later report. Desired state becomes stopped or removed. The server's AgentCanReportTaskStatus guard must not let an old running observation resurrect work after that desired transition. The focused control-plane tests explicitly cover nonresurrection and terminal duplicate idempotence. A lifecycle rejection does not physically stop a partitioned old container; it prevents an invalid state update in the authoritative record.

Cancellation stops the main timer loop, and graceful shutdown attempts a final heartbeat using a new background context bounded by GracefulShutdownTTL. Failure to notify shutdown is logged but does not

block termination forever. This is a best-effort notification, not proof that every container stopped or that the control plane observed shutdown before the process exited.

A subtle operational implication follows from sequential polling. If an external call ignores cancellation, cancellation of the top-level context alone does not prove every subordinate activity has ended. The supplied HTTP client has a 15-second timeout and request contexts, and runtime methods accept contexts, but each adapter's behavior still deserves its own test. Chapter 16 examines the log stream's additional goroutine and reader boundaries.

Development History and Atlas Lab

5cf9213849164ce1bebc4e042cc37bf5553c2427 introduced registration/heartbeat flow. 239eb6abab67773e20a18944ac38e2589802946a connected task assignment reconciliation. 4f25e78c229ba040c07a84801490485b5a0cbb2e changed credential reuse and agent authentication. Original prompts 07,08 and 19 are retrospective requested scope; the actual patches establish implementation history without proving a literal prompt execution transcript.

The source-only experiment runs agent restart/re-report, server heartbeat transitions, stopped-task nonresurrection and terminal duplicate cases. It uses injected clients/FakeRuntime and therefore does not need Docker or a database. The focused record passed with the exact package selection and test names in its raw output.

```
go test ./internal/agent ./internal/controlplane ./internal/auth \
  -run 'TestReconcileAssignedTaskReReports|
TestAgentTaskStatusCannotResurrect|
TestAgentTaskStatusTerminalDuplicate|TestAgentHeartbeat|
TestAgentCredential' \
  -count=1 -v
```

Run this from the disposable pinned source with Go 1.25, the environment contract's GOROOT/caches and external integration flags unset. Examine the test's retained objects: agent reconstruction keeps the server task and runtime container fixture. Record that scope before explaining its success. A separate lost-response teaching fixture should inject failure after server acceptance, not merely return an error before any mutation; those two cases exercise different ambiguity windows.

For an optional process experiment, use one owned server and agent on loopback, save their PIDs and exact task/container IDs, then terminate only the agent. Observe the returned node and task reports after restart. Do not stop host networking to imitate a partition, and do not share the worker node identity with unrelated containers. Stop the agent before deleting its exact owned runtime objects. A fresh server restart remains the separate state-loss experiment from chapter 6.

A heartbeat and a task report should also be tested under different failure schedules. Suppose the heartbeat succeeds while reporting running fails. The server can reasonably consider the node reachable while its task state remains stale. Conversely, a workload may keep serving traffic while heartbeats are lost. Observing one channel cannot substitute for inspecting the other. In a fake-client experiment, record the method, node/task identity and error for each call; then compare the server-side task snapshot after the next reconciliation pass.

This distinction matters for an incident dashboard. A green node icon does not establish that every task transition arrived, and a stale task icon does not establish that the local process has stopped. Preserve the last accepted heartbeat time, last accepted task report and a separately scoped runtime observation. They answer three different questions. The existing stale-report guards protect accepted control-plane state; they do not remotely interrupt a process on a partitioned host. An operator deciding whether to start a replacement must understand that additional physical fencing or externally enforced ownership is still a separate requirement.

Exercises

1. Identify which operations use the generic retry helper and which status-report operation does not. How can the latter still recover later?
2. Model a running report accepted by the server but not acknowledged to the agent. Which persistent identities make the next pass safe?
3. Explain why a rejected stale running report and an offline-node replacement do not establish physical fencing.
4. Design a cancellation test that distinguishes leaving Runner.Run from ending every transport/runtime activity. Include one observable resource that must close.

Worked answers retain the distinction between authentication, delivery and physical side effects. Use the protocol to reason about bounded trust and recovery; do not promote a successful heartbeat or fake restart into a broader guarantee than its actual path supports.

Chapter 14 – Health Checks That Respect Trust Boundaries

Atlas API's container is running, yet requests to its application path fail. Treating running as healthy would send traffic to a process that cannot satisfy users and could advance a rollout too early. A probe gives the agent another observation, but it also makes the agent initiate network requests on someone else's behalf. Correctness and outbound trust must therefore be examined together.

This chapter assumes agent reporting and published-port handling. You will trace a configured path into the exact HTTP/TCP target, calculate when repeated results cross a threshold and distinguish a bounded health check from general network isolation. The examples use fake checkers and owned loopback listeners; none contact cloud metadata services or arbitrary private hosts.

Running Is Not Healthy

Execution status and health answer different questions. Docker running means the process has started and has not yet been observed exited. An HTTP health result describes one request to one endpoint at one instant. Neither proves a complete business transaction, dependency availability or future responsiveness. Keeping these observations separate lets operators diagnose whether failure occurred before startup or after an application became reachable.

The agent only probes tasks whose actual state is running, healthy or unhealthy and whose healthcheck type is enabled. No configured check produces no new health report from this path. That absence should not be displayed as measured application health. A rollout or discovery component must interpret the task state it actually receives, rather than inventing a successful probe because a container exists.

A task can move healthy→unhealthy and later unhealthy→healthy according to the shared lifecycle guard. The server may also apply restart policy after unhealthy observations. Those downstream decisions must be traced independently: a probe returns an observation; it does not itself stop Docker, replace a node or select a rollout version. Claim ch14.health-state.

For Atlas, start from a path that serves successfully, then use the deliberately absent /atlas-deliberately-absent path in the lab fixture. Both manifests use the same image and service identity. The failed request isolates application-path health from image pull and process startup. Changing the image to a nonexistent reference would test a different failure phase and would not establish health threshold behavior.

HTTP and TCP Probe Decisions

healthProbe does not accept an arbitrary hostname from deployment YAML. It finds a suitable assigned TCP published port, builds the target at127.0.0.1 and combines the validated scheme/path for HTTP. A requested health port can match an assigned public or container port; an unassigned port does not yield a probe. Claim ch14.target.

internal/agent/agent.go, lines 562–585 at the locked revision.

```
        port, ok := healthcheckProbePort(check.Port, ports)
        if !ok {
                return health.Check{}, false
        }
        timeout := check.Timeout
        if timeout <= 0 {
                timeout = 2 * time.Second
        }
        target := "127.0.0.1:" + strconv.Itoa(port)
        switch check.Type {
        case types.HealthcheckHTTP:
                path := check.Path
                if !strings.HasPrefix(path, "/") {
                        path = "/" + path
                }
                scheme :=
strings.ToLower(strings.TrimSpace(check.Scheme))
                if scheme == "" {
                        scheme = "http"
                }
                return health.Check{Type: health.HTTPProbe, Target:
scheme + "://" + target + path, Timeout: timeout}, true
        case types.HealthcheckTCP:
                return health.Check{Type: health.TCPProbe, Target:
target, Timeout: timeout}, true
        default:
                return health.Check{}, false
```

This is a host-side probe of the task's published port, not a request inside the container's network namespace. A process bound on an unexpected port, a missing host binding or a wrong path can all make a running container unhealthy. If no assigned port matches, the agent skips probing; it does not magically discover an internal container address. That behavior is a useful source-reading checkpoint when a manifest looks valid but no health observations appear.

The domain/CLI validation requires a safe path and supported scheme. Absolute URLs and protocol-relative path forms must not turn a relative application path into a different target. The agent's target builder lowercases/defaults the scheme and ensures a leading slash, while the

validation layer establishes stronger permitted-input constraints. A builder's small normalization step should not be mistaken for a complete independent parser.

DefaultChecker dispatches HTTP and TCP probes. HTTP creates a GET request with a context and timeout, follows only permitted redirects, drains at most64KiB of response body and evaluates status. Statuses from200 up to but excluding400 are considered healthy after the client's redirect behavior. TCP merely establishes and closes a connection. A successful TCP connect does not prove the HTTP route or response body is correct.

The response body cap limits how much content is read; it does not reject a large response purely because more bytes exist. The huge-body test checks bounded reading rather than an invented payload-validation policy. This distinction prevents a security improvement from being oversold as a content guarantee. Application-specific health semantics, such as checking a JSON field, would need a separate supported checker or teaching extension.

Timeouts and Repeated Health Reports

Each check has a timeout, defaulting to two seconds when unset or nonpositive at the checker boundary. HTTP uses both a request context and client timeout; TCP uses a context-aware dial. A connection refusal, timeout or blocked redirect generally becomes a result with Healthy false and a message, while malformed configuration or an unsupported probe type can return an error. The agent handles those categories differently.

For an ordinary unhealthy result, the agent increments consecutive failures and clears successes. For a healthy result, it increments successes and clears failures. These are in-memory agent counters keyed by task identity. They are not persisted failure history and do not survive arbitrary agent reconstruction. Claim ch14.threshold.

When successes reach HealthyThreshold, the agent reports healthy if the task is not already healthy. When failures reach UnhealthyThreshold, it reports unhealthy with the probe message. Defaults are one success and three failures when the relevant threshold is nonpositive. A returned checker error increments the failure metric but returns early before the normal threshold-counter update. Do not count every possible checker error as one accumulated unhealthy observation without following that branch.

Consider Atlas configured with healthy threshold one and unhealthy threshold two. The first successful observation can report healthy. One later404 resets successes and records one failure but does not cross the unhealthy threshold. A second consecutive404 reports unhealthy. A subsequent success resets failures and can report healthy again, unless a server-side restart/deletion transition has already changed the task's admissible state.

Moment	Observation	Agent counter effect	Possible report
t0	Container starts	No probe result yet	Running
t1	HTTP200	successes1, failures0	Healthy
t2	HTTP404	successes0, failures1	None at threshold 2
t3	HTTP404	failures2	Unhealthy
t4	HTTP200	successes1, failures0	Healthy if lifecycle still allows

These labels are a logical timeline, not measured elapsed seconds. The loop schedules probes alongside heartbeat/task reconciliation and adds health jitter; it does not run a perfectly periodic independent probe every configured interval. Slow requests and other task work can affect observation timing. A threshold of two means two counted observations, not a promise of failure detection within exactly twice the interval.

SSRF Protection With Explicit Limits

The SSRF hardening is a chain of controls: safe scheme/path input, assigned-port selection, a loopback-generated initial target and same-origin redirects. sameOriginRedirect compares lowercased scheme and host, including the port, against the original URL. It also bounds the redirect chain. A redirect to another loopback port is still another origin and is blocked. Claim ch14.redirect.

internal/health/health.go, lines 97–110 at the locked revision.

```
func sameOriginRedirect(origin *url.URL) func(*http.Request,
[]*http.Request) error {
        originScheme := strings.ToLower(origin.Scheme)
        originHost := strings.ToLower(origin.Host)
        return func(req *http.Request, via []*http.Request) error {
                if len(via) >= 10 {
                        return fmt.Errorf("stopped after 10
redirects")
                }
                if strings.ToLower(req.URL.Scheme) != originScheme
|| strings.ToLower(req.URL.Host) != originHost {
                        return fmt.Errorf("healthcheck redirect
blocked: %s", req.URL.Redacted())
                }
                return nil
        }
}
```

Use owned local handlers to test this. Handler A can redirect to handler B on a different port; the result should be unhealthy and B should not be treated as the approved target. A same-origin redirect can reach another path on A. There is no need to contact a real metadata address to establish that cross-origin behavior. A fake HTTP transport is also appropriate when testing request construction without listeners.

The generic DefaultChecker accepts a Target string and is not itself a universal address allowlist. The safe initial target described here comes from the agent's builder. Another caller using the checker directly would

need its own trusted-target policy. This is why source claims must include both the package and its current caller rather than attributing all target protection to one function.

Loopback targeting is also not tenant network isolation. The agent runs on a host where other processes can listen, and a permitted published port must correspond to the intended container binding. Host-network mode and daemon access introduce broader risks that a redirect check cannot remove. The book's lab avoids arbitrary host probing and keeps workload exposure explicitly separate from operator-facing loopback listeners.

History and Atlas Lab

Task probing was introduced in ee067bd0f02566eb600031a0cb81f25919d52445. The later hardening commit f92f0a4fcfd760cbc9f850ab4c2e3e6d5dc8a081 changes scheme handling, validation, redirects and body reading. Original prompt 11 is retrospective intent; prompt 53 requests the SSRF work. The inspected tests and changed hunks define the actual controls, not a blanket claim that the agent cannot make any dangerous network request.

The experiment needs Go 1.25 and permission for temporary owned loopback listeners. Docker and PostgreSQL are unnecessary. The first attempt in the restricted environment failed because opening the test listener was denied; that failure record is preserved. The same pinned tests then ran with the approved loopback capability and passed. This is an environment restriction followed by an actual successful run, not a source repair.

```
go test ./internal/health ./internal/agent \
  -run 'TestHTTPHealthCheck|TestTCPHealthCheck|
TestHealthProbeRequiresAssignedPort|
TestReconcileAssignedTaskReportsHealth' \
  -count=1 -v
```

Execute from the disposable source copy with the environment contract's toolchain/caches. Inspect the raw output for healthy/unhealthy HTTP cases, timeout, same-origin and external redirect cases, huge-body

bound, TCP behavior, assigned-port construction and agent thresholds. The allowed external-redirect test uses local fixtures; no real metadata endpoint is contacted.

All listeners belong to httptest or the local test fixture and close when the tests complete. No system networking configuration is changed. If adding an Atlas process experiment, use the owned unhealthy manifest, record actual conditions and restore the healthy fixture afterwards. Report timings only if measured; preserve image, task and probe configuration with the observation. Stop the exact owned agent/server before removing container IDs.

Choose the probe endpoint as deliberately as its timeout. An Atlas root page can return200 from a cached handler even while the dependency required by the API is unavailable. A readiness endpoint can instead check the minimum dependency needed to accept useful traffic, but it should avoid expensive or destructive work. If every probe performs a costly database operation, the health mechanism can amplify an outage. These are application-level design choices; the orchestrator sees only the resulting observation.

A safe local experiment can expose the distinction with two handlers in an owned loopback server. Keep one endpoint returning200 and make the other return503 under an injected dependency flag. Predict the resulting counters for each chosen path before sending a probe. This exercise requires no real database and does not justify probing a remote service. Record which path was configured and the exact response sequence, because otherwise a healthy result says little about what was actually tested. When the dependency recovers, require the configured consecutive success threshold before expecting a healthy report. Counter reset behavior, endpoint semantics and downstream replacement policy must all agree before an operator can interpret recovery accurately.

Exercises

1. Starting from running with both healthy and unhealthy thresholds set to two, predict reports for 200,404,200,404,404. Explain why total failures are not the deciding count.
2. Why does a TCP success not establish HTTP health? Give an Atlas failure that separates them.
3. Compare a redirect to another path on the same origin with a redirect to another port on127.0.0.1. Which source check decides?
4. An engineer calls DefaultChecker directly with an arbitrary Target and cites this chapter as proof of an address allowlist. Identify the missing caller boundary and propose a safe local test.

Worked answers retain each counter transition and trust boundary. Choose health checks for an explicit observation, preserve bounded network behavior, and avoid presenting successful probing as evidence of complete availability or host isolation.

Chapter 15 – Events, Audits, and Notification Boundaries

Atlas API changed replicas, a task failed and an operator wants to know what happened. Three different records may answer parts of that question: a domain event says a change occurred, a diagnostic log explains a code path, and an audit record attributes an operation to an actor. A webhook adds a fourth concern: delivering a selected observation to another system. Treating all four as one durable stream would hide their different storage and failure semantics.

This chapter assumes the service lifecycle and transaction discussion. You will reconstruct an actual in-process Atlas event sequence, trace actor/redaction fields and examine a notification retry without claiming an exactly-once external effect. Default MemoryService history remains process-local. The original durable-events prompt title is historical requested scope, not proof that the default binary persists its event history.

Events, Diagnostic Logs, and Audits

Domain events have an ID, namespace, type, severity, source, message, related-object identity and timestamp. Their purpose is to connect observations to service/task/node changes. Diagnostic logs instead describe the executing code's behavior, such as a failed Engine call or retry. An audit record adds actor type/ID, action, target, request ID, source IP, outcome and redacted metadata. Claim ch15.record-types.

A single Atlas scale request can therefore produce a service.scaled event and an audit operation while also generating diagnostic logs. Their identifiers and ordering are related only where the implementation explicitly connects them. An event's source string agent is not the same thing as a cryptographically attributable audit identity. A log line containing a service name is not a durable mutation receipt.

The teaching fixture used the real MemoryService methods to register a node, create atlas-api, accept pulling/created/running/healthy reports and scale to two. It did not execute a container: the report source is explicitly injected, and its container ID says fixture-container-no-docker. The actual event capture contains seven events with generated IDs and UTC timestamps. Claim ch15.timeline.

Read that capture in chronological order because ListEvents returns the newest observations first. It contains node.registered, service.created, four task.status events and service.scaled. The task.status messages are generic; their event text alone does not encode every reported lifecycle value. The fixture's request sequence supplies that additional context. This is a useful limit on reconstructing a deployment exclusively from human-readable messages.

Record	Primary question	Example scope
Domain event	What state-related observation occurred?	service.scaled related to a service ID
Diagnostic log	What did this execution path encounter?	Runtime error while polling a task
Audit	Who attempted which operation with what outcome?	Actor, action, target and request metadata
Notification delivery	Was a selected event sent to a sink?	HTTP attempts and final response

The actual process/Docker deployment has separate evidence in the book's runtime lab. Do not substitute the in-process timeline for proof of image pull, network reachability or workload execution. The value of this fixture is that it exercises the real event-generation methods with a tightly controlled sequence while keeping those physical effects out of scope.

Attribution, Redaction, and History

The API's recordAudit path derives an actor, constructs a Log and appends it through the configured audit store. An authenticated principal supplies a user subject. Agent routes can use node metadata, while absent identity has explicit fallback values such as unknown or anonymous. A fallback value is a signal of limited attribution, not proof a named human performed the request.

Request ID and source IP help correlate observations, but their trust depends on how the request reached the server. The code accepts forwarded metadata; a deployment that exposes the API directly must not assume every forwarded value came from a trusted reverse proxy. Correlation is useful even when a field is untrusted, provided it is not silently promoted to an authenticated identity claim.

Audit.Normalize trims fields, supplies defaults, normalizes timestamps to UTC and calls RedactMetadata. Redaction inspects metadata keys for markers such as password, token, secret, authorization and credential. URL metadata can have embedded user information replaced. This is a concrete heuristic with a known input shape, not a universal parser for all secrets in arbitrary prose.

internal/audit/audit.go, lines 92–107 at the locked revision.

```
func redactMetadataValue(key string, value string) string {
        if isSensitiveMetadataKey(key) {
                return "[REDACTED]"
        }
        if strings.Contains(strings.ToLower(key), "url") {
                parsed, err := url.Parse(value)
                if err == nil && parsed.Scheme != "" &&
parsed.Host != "" && parsed.User != nil {
                        parsed.User = url.User("[REDACTED]")
                        return parsed.String()
                }
        }
        return value
}

func isSensitiveMetadataKey(key string) bool {
        normalized :=
strings.ToLower(strings.ReplaceAll(strings.TrimSpace(key), "-",
"_"))
```

If a secret appears under a harmless-looking key or in an unrecognized message format, a key-based rule may not find it. Conversely, a key containing a sensitive substring can be redacted even when the value is not secret. Prefer constructing minimal metadata at the source rather than treating the sanitizer as permission to include entire request bodies, environment maps or raw credentials. Claim ch15.redaction.

The API audit append is best effort in the inspected path: failure is logged rather than automatically rolling back the completed user operation. That distinction differs from transaction-scoped strict domain events used by some controllers. An operator can receive successful mutation behavior while the audit write fails. If a future requirement demands every privileged mutation have an atomic durable audit record, it needs an explicit transaction/operation design.

The domain events helper also makes its mode visible. Emit defaults missing timestamps/severity and attempts AppendEvent. In best-effort mode, append failure is logged and Emit returns nil. With Strict, the append error is returned so a transactional caller can abort. Strict

emission without a transactional enclosing mutation is still not a complete atomicity guarantee: the mutation may already have occurred elsewhere.

Notification Delivery as an External Effect

The notifier has an explicit set of event types eligible for delivery. Dispatcher.Notify ignores unrelated types, loads sinks and sends to them sequentially. Sender.Deliver validates a sink, redacts the event, builds a JSON Delivery payload and performs HTTP POST attempts. Default settings are three attempts, 100 millisecond base exponential backoff and a client timeout. Claim ch15.delivery.

internal/notifications/notifications.go, lines 86–105 at the locked revision.

```
        var last error
        for attempt := 0; attempt < s.attempts; attempt++ {
                req, err := http.NewRequestWithContext(ctx,
http.MethodPost, sink.URL, bytes.NewReader(payload))
                if err != nil {
                        return err
                }
                req.Header.Set("Content-Type", "application/json")
                req.Header.Set("User-Agent", "orch-notifier/1")
                if sink.SigningSecret != "" {
                        mac := hmac.New(sha256.New,
[]byte(sink.SigningSecret))
                        _, _ = mac.Write(payload)
                        req.Header.Set("X-Orch-Signature",
"sha256="+hex.EncodeToString(mac.Sum(nil)))
                }
                resp, err := s.client.Do(req)
                if err == nil {
                        _, _ = io.Copy(io.Discard, resp.Body)
                        _ = resp.Body.Close()
                        if resp.StatusCode >= 200 && resp.StatusCode
< 300 {
                                return nil
                        }
```

When a signing secret exists, the sender attaches an HMAC-SHA256 signature over the payload. That lets a correctly configured receiver verify payload authenticity relative to the shared key. It does not encrypt the payload, authenticate every downstream workflow or prevent the receiver from processing the same event twice. The payload contains a timestamp and event identity that can support receiver-side deduplication, but no exactly-once receiver behavior is implemented here.

The failure window is an HTTP response lost after the receiver accepted the request. The sender sees a transport failure and retries; the receiver may now observe the same notification more than once. A two-hundred response establishes acceptance by that HTTP endpoint, not successful completion of whatever incident workflow it triggers. Reliable external automation should be designed around idempotent event identity and explicit handling of duplicates.

The actual focused notifier fixture uses a fake RoundTripper. It returns service-unavailable twice, then success, captures the body and signature and checks that selected secret values are absent. The second fixture exercises delivery failure and the dispatcher event filter. These tests passed. They do not send a real webhook, validate arbitrary receiver software or prove every possible secret representation is redacted.

A separate integration concern is how notifications are invoked. MemoryService appends an event to its in-memory slice, then launches a goroutine calling Notify with context.Background and discards its returned error. The user mutation does not wait for delivery, which isolates its immediate latency from the sink. It also means there is no durable delivery queue, retry-after-process-restart record or guaranteed shutdown drain in that path.

Repeated failures can consume goroutines and outbound requests while the process remains alive; losing the process loses those transient delivery activities. A teaching outbox design would store the event and a

delivery intent transactionally, then process bounded attempts with deduplication and retention. That is a prospective extension, not a capability inferred from the current Sender type.

When History Is Not Durable

The default server's event and audit collections belong to MemoryService. Retention settings can remove older entries from those collections, but they do not persist them. A fresh server process starts new maps/slices. The PostgreSQL store and event tables exist as implemented packages, yet their existence does not redirect the running application's writes. Claim ch15.durability.

This changes the operational interpretation of an empty history page. It may mean no matching events, a filter mismatch, retention or process state loss. An API response containing zero events is not proof no operation ever happened. Preserve critical lab output before intentionally restarting or deleting the service; use UTC timestamps and IDs to distinguish a fresh run from an earlier one.

Events and audits also have different visibility permissions and filters. A service filter is useful for a timeline but can omit node-level observations relevant to a task failure. Audit access requires its actual API authorization, not the same assumption used for reading application logs. Build an incident bundle deliberately: service/task/node state, relevant domain events, authorized audit metadata and diagnostic/runtime logs each contribute different evidence.

History begins with 7ff58cef0f736ac9076942ce3ef1039d1fe72c72, which adds event APIs/helpers and related fields. 7935c334dcd264e70abb9e795889010b9ecdb03b adds audit attribution/redaction paths. 4d641bbe8913da568d6d9f4697a0144314e10b87 batches notification work with jobs, storage and DNS. Do not invent a separate commit for every late development prompt. Original13 is retrospective intent;46 and 80 identify later audit/notification requests.

Atlas Lab: Capture a Timeline and Test a Sink

The safe lab needs Go 1.25 and the disposable source copy, with no Docker or database. Copy examples/teaching-extensions/ch15/main.go into book-events beneath that module, then run it using the pinned toolchain and run-owned caches. The fixture asserts that service creation, scaling and task-status events exist and emits the real event fields as JSON.

```
go run ./book-events
go test ./internal/events ./internal/notifications \
  -run 'TestEmit|TestDelivery' -count=1 -v
```

The actual fixture had an initial compile error in its teaching registration field name; that raw attempt is preserved separately. The corrected fixture uses the real NodeRegistration.Name field and passed. No upstream source changed. This is why the reproducible artifact is the corrected canonical fixture plus its named execution record, not a rewritten claim that the first attempt worked.

Inspect event IDs and related-object IDs, reverse the returned order for a chronological display and identify where generic messages need request context. Then examine the fake sink's three attempts, signature prefix and redacted payload assertions. The fake transport avoids external requests and cannot change a real incident-management account. Cleanup is ordinary process exit and optional removal of the copied teaching directory.

For an optional live Atlas capture, run events with an explicit server, namespace and service filter, save the actual output before cleanup and keep node-level history separately. Never include bearer tokens or secret request values. A local HTTP sink should bind loopback, be owned by the run and use disposable signing material. Record its received event IDs to detect duplicates rather than assuming a successful sender return means unique processing.

Exercises

1. Why do four generic task.status events not independently prove four specific lifecycle values? Which additional record supplies that meaning?
2. Compare best-effort Emit, strict Emit and a strict event inside a database transaction. Which failure can roll back the mutation?
3. Explain how a notifier can deliver twice after only one event was appended. What receiver key should be used for deduplication?
4. An operator restarts the default server and sees an empty event list. Enumerate plausible interpretations and the evidence needed to distinguish them.

Worked answers connect record shape, atomicity and delivery scope. Preserve separate evidence channels and name their storage lifetime before using history as an operational guarantee.

Chapter 16 – Log Streaming Without Losing Control

Atlas API is running, but an operator needs to see what it wrote before a failed request. The CLI opens a stream through the server, which contacts the worker, which reads Docker logs. Each hop owns a connection or reader, and the operator can disconnect at any moment. A useful log feature must therefore do more than print lines: it must identify the right task, preserve meaningful error boundaries and stop work when its consumer disappears.

This chapter assumes agent identity, Docker runtime methods and the difference between events and application output. You will trace the actual files, distinguish stdout/stderr framing from the text the operator receives, and reason about slow consumers and cancellation. This path is an on-demand stream, not an invented persistent centralized collector.

The Client-to-Server-to-Agent Stream

The command is logs <service-name-or-id>, with optional --task, --follow and --tail. The CLI resolves the service before requesting its log stream; selecting a task makes the target explicit during a rollout or scale change. The client forwards a request to /v1/logs. These are current CLI shapes, not commands inferred from a planned package named logs.

On the server, the handler resolves a task, requires a known container ID and obtains its node. Offline or unknown node state is rejected before contacting the worker. The server sets a plain-text response and passes the task ID, advertised agent address, follow/tail values and configured bootstrap token to AgentHTTPLogStreamer. Claim ch16.proxy.

The streamer builds /v1/agent/logs with URL-encoded query parameters, attaches the supplied bearer token and creates the outbound request with the incoming context. A non-success agent status becomes an error

with a bounded response-body read. Successful response bytes are copied to the operator's writer. The public stream therefore depends on both API authorization and the separate server-to-agent credential path.

internal/api/logs.go, lines 33–51 at the locked revision.

```
        baseURL := normalizeAgentURL(req.AgentURL)
        values := url.Values{}
        values.Set("task_id", req.TaskID)
        if req.Follow {
                values.Set("follow", "true")
        }
        if strings.TrimSpace(req.Tail) != "" {
                values.Set("tail", strings.TrimSpace(req.Tail))
        }
        httpReq, err := http.NewRequestWithContext(ctx,
http.MethodGet, baseURL+"/v1/agent/logs?"+values.Encode(), nil)
        if err != nil {
                return fmt.Errorf("create agent logs request: %w",
err)
        }
        if req.Token != "" {
                httpReq.Header.Set("Authorization", "Bearer
"+req.Token)
        }
        resp, err := client.Do(httpReq)
        if err != nil {
                return fmt.Errorf("request agent logs: %w", err)
```

The worker's log handler accepts its configured token, finds a managed container by task ID and calls Runtime.StreamLogs. At this baseline the worker log handler uses the bootstrap token configured at startup; it is not automatically the same short-lived enrolled credential used for node reports. Conflating those credentials would conceal an important operational trust boundary.

An advertised agent address is also an outbound destination used by the control plane. Treat worker enrollment and allowed worker endpoints as part of trusted configuration. The health-check target restrictions from chapter 14 apply to their own caller path; they do not automatically constrain the server's log proxy. Each outbound facility needs a separate source trace and network policy.

Docker Frames and Application Output

Docker can multiplex stdout and stderr in its log transport. The Engine adapter obtains a reader and uses stdcopy.StdCopy to split it into stdout and stderr pipes. Separate scanner goroutines turn each pipe into LogLine values tagged with Stream and a read-time UTC timestamp. This structured intermediate representation carries information that the final plain-text handler does not preserve. Claim ch16.frames.

internal/docker/docker.go, lines 326–335 at the locked revision.

```
                stdoutReader, stdoutWriter := io.Pipe()
                stderrReader, stderrWriter := io.Pipe()
                copyErr := make(chan error, 1)
                go func() {
                        _, err := stdcopy.StdCopy(stdoutWriter,
stderrWriter, reader)
                        _ = stdoutWriter.Close()
                        _ = stderrWriter.Close()
                        copyErr <- err
                }()
```

The agent sends line.Line with Fprintln and flushes when the writer supports flushing. It discards the LogLine.Stream field and timestamp in that response. The operator sees newline-delimited text, not a structured record that reliably identifies stderr versus stdout. A displayed timestamp should not be invented from the intermediate read time when the output did not include it.

The two scanner goroutines also do not establish one globally ordered sequence across both streams. Each stream retains its reader's order, but their shared-channel interleaving depends on scheduling and arrival. If precise cross-stream ordering matters, the application should emit its own correlation/timing fields or a future structured transport should expose an explicitly defined ordering model. The current text output cannot reconstruct information that was discarded.

Application stdout/stderr differs from orchestration domain events. A task.failed event explains a control-plane observation and related task identity; an application error line explains the workload's own execution. A deployment can generate domain events without ever starting a container, and a healthy container can write errors without changing service intent. Combine both records during diagnosis rather than searching one as a substitute for the other.

The scanner uses its default buffer behavior without a custom Buffer configuration in this source. Exceptionally large unbroken lines can therefore surface scanner errors. That is a bounded parsing concern, not an unlimited-log guarantee. A teaching structured-log extension should choose and test its line/frame limits explicitly, including what the operator sees after the limit is exceeded.

Tail, Follow, Disconnect, and Cancellation

Tail selects how much prior output Docker should return, while follow keeps reading later output. The CLI passes a string; the worker's tailValue accepts integer text and otherwise falls back to all. This is permissive normalization rather than strict validation of every nonnegative bound. A malformed value can therefore request more history than the operator intended, and a negative integer is left for downstream behavior.

Runtime.StreamLogs forwards follow, tail and optional runtime fields such as since/until/timestamps to the Engine request. The public CLI discussed here exposes only its current subset. Do not document unimplemented CLI switches merely because the lower-level LogOptions structure has those fields. Claim ch16.options.

The handler selects between incoming lines and request-context cancellation. If the operator disconnects and the request context is canceled, it can return promptly. The outbound proxy request also carries context, so cancellation can propagate to the worker connection. At the runtime layer, scanLogStream checks cancellation before sending a scanned line to the channel.

internal/docker/docker.go, lines 612–623 at the locked revision.

```
func scanLogStream(ctx context.Context, reader io.Reader, stream
string, out chan<- LogLine, done chan<- error) {
        scanner := bufio.NewScanner(reader)
        for scanner.Scan() {
                select {
                case <-ctx.Done():
                        done <- ctx.Err()
                        return
                case out <- LogLine{Stream: stream, Line:
scanner.Text(), Timestamp: time.Now().UTC()}:
                }
        }
        done <- scanner.Err()
}
```

Those checks are useful but not identical to proving every reader and goroutine closes under every blocked-I/O state. A scanner can be waiting in reader.Scan before it reaches the select, and a copy goroutine can be blocked on a pipe. Engine StreamLogs coordinates both scanner completions and the framing copy before its deferred reader close. A complete transport shutdown test must exercise those blocked states, not only a cooperative fake that immediately notices context cancellation.

This is the chapter's failure window. The client disappears while one stream writer or reader is blocked. The HTTP handler may return, yet a subordinate goroutine can remain if the cancellation path does not unblock its specific I/O operation. Source inspection identifies the risk and the tests establish their actual scope; neither permits a blanket assertion that all possible real Engine streams are leak-free.

Slow consumers create related backpressure. The adapter's line channel is unbuffered, and the HTTP writer can block while the operator reads slowly. Buffering everything would merely move the problem into memory growth. A deliberate design chooses a bounded queue, a cancellation/deadline strategy and an observable error/drop policy. That would be a separately tested teaching extension, not a configuration switch already present in this stream.

Transport errors after streaming starts have another limit. The server flushes headers before copying worker output. If the worker later fails, the server cannot replace the already-started plain-text response with a fresh structured JSON error status. It logs the failure and the client can receive a truncated stream. A successful initial HTTP status is therefore not a receipt that the complete requested history arrived.

Failure point	Operator-visible possibility	Additional evidence
No task container known	Request rejected before streaming	Task/container state
Worker returns non-success	Proxy logs error after headers may start	Agent/server diagnostic logs
Client disconnects	Stream ends/cancels	Handler and runtime cleanup assertions
Docker disappears mid-follow	Partial output, transport error	Runtime reader/copy error
Task/container deleted	Existing reader may end or fail	Exact task/container IDs and deletion timeline

Boundaries of Log Retention

The stream reads whatever the configured Docker logging path can supply. The orchestrator does not persist a duplicate central log archive in MemoryService, PostgresStore or an external collector through this implementation. Removing a container or changing daemon logging/rotation can change what remains available. An older task's log request cannot be guaranteed after its runtime object disappears.

A service name is particularly ambiguous during a rollout. Multiple tasks may exist across versions, and a default task selection can differ from the operator's failing replica. Record service ID, task ID, container ID, node ID and deployment version when saving a diagnostic excerpt. Use --task when reproducing a specific failure. A screenshot of generic service logs without those identities is weak evidence about which version actually emitted a line.

The text stream also does not universally redact application output. The workload can print a password or selected user value, and the handler writes that text. Audit metadata redaction and notifier sanitization are different code paths. Use deliberately nonsecret fixtures, redact copied evidence where required and keep the raw sensitive data out of the manuscript rather than claiming a sanitizer that is absent.

For a centralized retention design, define collector ownership, authentication, bounded storage, rotation, tenant scoping, correlation and restart behavior. Test the failure case where collection stops while the workload continues. Calling the existing log proxy a collector would skip all of those contracts and create an operational expectation the source does not meet.

History and Atlas Lab

39e2359653e0e8790feda86eb5fbee3eb5ae509f introduced the agent HTTP handler, API proxy and basic streaming integration. Its patch also adds cancellation-related test structure. Original development prompt 12 is retrospective intent, not a timestamped claim that the described test ran when that prompt first appeared. Follow the actual internal/agent, internal/api and internal/docker files rather than the planned package tree.

The source-only lab uses a FakeRuntime whose log stream intentionally blocks until cancellation. TestLogHandlerStopsOnCancellation starts the handler with a request context, waits until streaming begins, cancels and requires handler completion. The runtime's own fake-stream fixture drains its channels after cancellation. The actual focused run passed both. It does not exercise a real Engine reader blocked in stdcopy or every proxy hop.

```
go test ./internal/agent ./internal/docker \
  -run 'TestLogHandlerStopsOnCancellation|
TestFakeRuntimeStreamLogsCanBlockUntilContextCancellation' \
  -count=1 -v
```

Run from the disposable pinned source with Go 1.25 and the environment contract's GOROOT/cache settings. The fixtures create no host container and require no PostgreSQL or external network. Their cleanup is context cancellation, goroutine completion and process exit. Reading the completion assertion is essential: a test that only calls cancel without waiting would establish almost nothing about cleanup.

The separate actual Docker flow verifies deploy, HTTP access, bounded logs, scaling and deletion for the owned lab workload. That proves a useful end-to-end path in its environment, but it is still not the targeted blocked-stream disconnect experiment. Keep those claims separate. An optional real follow/disconnect test should record reader/goroutine completion and task identity while removing only the exact owned container after the agent is stopped.

Exercises

1. Which stdout/stderr information exists in LogLine but disappears in the final text response? Why can line order not recover it?
2. Compare tail and follow. What does a malformed tail string become in the worker handler?
3. Explain why the passing fake cancellation test does not prove every Engine pipe/scanner exits after a client disconnect.
4. Design a bounded slow-consumer extension with an explicit overflow/deadline policy and a test involving a disappearing task.

Worked answers separate the observed fake cleanup, the real bounded log path and the still narrower blocked-I/O question. Preserve that evidence budget when deciding what operators can safely expect from streaming and retention.

Chapter 17 – Rolling Updates and Rollback Invariants

Atlas API already has useful replicas. Its next image adds a feature, but the operator must change those replicas without confusing a deployment request with successful replacement. One new container can be created before it is ready, one old task can be told to stop before Docker acknowledges removal, and traffic can still use an endpoint cached before either change. A rolling update is therefore a sequence of observations and bounded decisions, not an atomic exchange of images.

This chapter assumes the task lifecycle, agent report path and health thresholds from Chapters 4, 13 and 14. You will calculate surge and availability bounds, distinguish failed target versions from previous successful versions, and identify conflicting user actions. The concrete mechanism is the rollout controller used by default server startup. Its state and history still reside in MemoryService; learning its count invariants does not remove the process-loss boundary established earlier.

Service Versions and Deployment Ownership

RolloutService changes the service's desired image, increments DeploymentVersion and saves a versioned specification before recording a pending deployment. FromVersion and ToVersion identify the transition. The API returns ownership of an operation that will advance later; it does not wait for every agent to start a container. The controller retrieves deployments in pending, running and rolling_back states, sorting by creation time and ID. This makes iteration order inspectable without making the collection durable.

A deployment is interpreted against current service replicas and tasks. Task.Version matters: an old image running under an old task belongs to a different cohort from a new target. A task from neither deployment

version is ignored by this particular accounting function. That is a reason to retain versioned evidence during triage rather than merely count every container with the service's name.

The source distinguishes active tasks, available old tasks and healthy new tasks. Old availability uses the lifecycle helper accepting running or healthy. New readiness is stricter: only actual healthy increments newHealthy. A new task whose container is merely running occupies active capacity but cannot authorize stopping an old replica. This difference is easy to lose when a dashboard compresses every live task into a green count.

Unchanged source excerpt — internal/rollout/rollout.go:376–384:

```
                if task.ActualStatus == types.TaskFailed {
                        state.newFailed++
                }
                if isActive(task) {
                        state.newActive++
                        if task.ActualStatus == types.TaskHealthy {
                                state.newHealthy++
                                state.available++
                        }
```

The claim here is bounded: this controller uses reported states from its store. It does not independently probe the application or synchronously update every proxy. A routing consumer must also observe the new endpoint set before client continuity can be evaluated. Its arithmetic can be correct while a client still reaches a stale or unreachable endpoint. Keep controller evidence and traffic observations separate throughout the experiment.

Max Surge and Max Unavailable Arithmetic

Take Atlas at three replicas, MaxSurge one and MaxUnavailable zero. The creation ceiling is three plus one: four active tasks. Initially three old tasks are available and there are no targets. One pass may create a target,

filling the fourth slot. Because that target is pending, no old task can stop. Repeating the pass without new reports cannot manufacture progress: the surge slot is already occupied and newHealthy remains zero.

When the first target reports healthy, the controller can stop an old task while keeping three available replicas. Its stop guard calculates the unavailable count after the prospective stop. With four available tasks, stopping one leaves three, so three minus three is zero. A second stop would leave two, making the unavailable count one and violating this deployment's zero allowance. The next pass uses desired/store accounting: a task whose desired status is stopped or removed no longer counts active, even before the agent acknowledges a physical stop. Capacity in this calculation can therefore be reused while its old container still runs.

Unchanged source excerpt — internal/rollout/rollout.go:244–252:

```
        for _, task := range oldTasks {
                if state.newHealthy <= state.oldStopped+stopped {
                        break
                }
                if service.Spec.Replicas-(state.available-stopped-1)
> deployment.MaxUnavailable {
                        break
                }
                transactional := c.transactional()
                stoppedTask := types.Task{}
```

There is another guard before that arithmetic: newHealthy must exceed oldStopped plus the stops planned in this pass. It relates each removed old replica to replacement readiness. Consequently MaxUnavailable is not a command to stop that many old tasks immediately. For example, a three-replica deployment with surge zero can stall because no target can be created and no healthy replacement exists to authorize stopping. The source's conservative ordering is more specific than a generic rolling-update policy description.

The creation loop tracks totalActive after each successful task write; stopping tracks the same local pass. Neither count turns a distributed runtime observation into a transaction over Docker. Desired stopped/removed excludes a task from active accounting before its actual state becomes terminal, and a stop directive can coexist with a container that has not received it. The controller also decrements its pass-local oldActive and totalActive immediately after a successful StopTask; it can declare rollout success while physical cleanup remains pending. Developers should identify whether a suspected limit violation concerns desired objects, reported task states or actual live processes before changing the arithmetic.

Atlas observation	Old active	New active	New healthy	Controller consequence
Before update	3	0	0	One surge slot can be filled
Target pending	3	1	0	Wait; no old stop authorized
Target healthy	3	1	1	One old stop can preserve availability
One old task desired stopped, actual stop unacknowledged	2 in controller accounting	1	1	Desired intent frees a count slot; physical container may remain live

This table is a source-derived scenario, not a captured execution trace. In a real run, attach the task IDs and timestamps that make each row meaningful.

Failed Targets and Previous Successful Versions

A target in TaskFailed causes the deployment to become failed. TaskUnhealthy is a different state: rolloutStateFor does not increment newFailed for it. The agent reports unhealthy after its threshold; MemoryService.ReportTaskStatus converts that report to failed when the service restart policy permits replacement. Without that conversion, an

unhealthy stored target is distinct from failed. The rollout controller does not interpret raw probe responses directly. During the intermediate window the target can occupy a surge slot while progress waits.

Imagine a healthy first target and then a second target whose application returns an error. The first target may already have allowed one old replica to stop. A later target failure marks the rollout failed; it does not atomically resurrect the stopped container. The remaining old and new cohorts are evidence of a partially advanced deployment. An operator needs their versions, states and endpoints before selecting recovery.

RollbackService finds a previous successful version and restores its specification as the desired target. It records a new deployment from the current version to the earlier one, preserving the failed deployment's history. TestRollbackAfterFailedRollout asserts version two returning to version one and confirms that the failed forward deployment remains failed. TestRollbackWhenNoPreviousVersionExists rejects a rollback without such a version. These are actual source assertions, not proof every runtime image remains pullable.

An in-flight rollback is idempotent: repeated calls return the existing operation rather than generate another reverse transition. That property is valuable when the response to the first request is lost. It is scoped to the same live control plane and operation identity. A new default server has neither the version history nor the prior operation maps, so retrying after process loss is a different situation.

The failed-image timeline should therefore name five events: forward request accepted, target created, target health degrades, deployment becomes failed, reverse deployment accepted. Add a sixth observation for reverse completion only when task reports establish it. Preserving the failed operation allows an incident report to explain why rollback was necessary instead of rewriting history into a smooth success story.

Locks Against Conflicting User Actions

Operation locks are enforced through service/deployment state under the memory service's mutex. An identical repeated rollout can return the active deployment. A different rollout conflicts, and rollback during an active forward rollout conflicts. These rules avoid two callers claiming incompatible desired versions at once. They do not constitute a distributed lock across independent servers.

Deletion has precedence. TestDeleteWinsOverActiveRollout requests delete, checks that the active deployment becomes failed, and rejects a later rollout against the deleting service. TestScaleAndDeleteOperationOrdering shows that scale after delete conflicts while scale before delete can complete before deletion wins. Repeated delete is idempotent. The practical rule is to classify an error as an operation conflict before retrying a different request; a blind retry can obscure the operator's intended order.

A failed deployment is terminal, so the operator can explicitly start rollback after failure. There is no need to invent a force flag merely because the original development prompt mentions one possible policy. Use the actual CLI and control-plane behavior. The lock tests are particularly useful because they make the ordering contract visible without requiring a timing-sensitive terminal demonstration.

Atlas Lab: Separate Count Invariants from Traffic

Use the disposable pinned source copy with Go 1.25 and an owned cache. Run the named deterministic tests first; they need no Docker or database. Their fakes explicitly mark tasks healthy, which is exactly why they can prove controller decisions reproducibly but cannot prove uninterrupted application traffic.

```
go test ./internal/rollout -run 'TestController(SuccessfulRollout|
FailsWhenNewTaskFails|RespectsMaxUnavailable|RespectsMaxSurge|
IdempotentResume|CompletesRollback)$' -count=1 -v
go test ./internal/controlplane -run 'Test(Rollback|
RolloutWhileRolloutInProgress|DeleteWinsOverActiveRollout|
ScaleAndDeleteOperationOrdering)' -count=1 -v
```

Expected observations are the assertions just described: a bounded target cohort, no old stop before replacement health, failed forward history preserved, an idempotent reverse operation and clear conflicts. The fresh package-test record includes the rollout/control-plane assertions with caching disabled. Its fakes establish model behavior, not continuous client availability. The rollout laboratory separately captures seven actual controller checkpoints; deployment success can precede acknowledgement of the final pending stop.

For a separate real Atlas run, record initial service version, all task/container IDs and direct HTTP responses. Use the case-study health-enabled manifest and an owned second image. Request rollout with one surge slot, then observe deployment status, task version/status and endpoint changes after each interval. Inject a failing target using the dedicated unhealthy manifest or nonexistent lab image; never modify a user service. Record both the failed deployment and the reverse operation.

Abort if the service name resolves outside the recorded run namespace, if expected old endpoints disappear before the recorded availability permits it, or if the selected failure affects an unrelated container. Cleanup deletes only Atlas resources created by that run and checks

residual container IDs. A passing count fixture cannot waive this cleanup or routing evidence. The engineered decision is to trust each observation for the boundary it actually measures.

Exercises

1. Atlas has three old available replicas, one healthy target, surge one and unavailable zero. How many old replicas may stop in the next decision, and which guard prevents the second stop?
2. Explain why a new running target cannot replace an old running replica in this controller even though both appear live.
3. A rollout has one healthy target and one unhealthy target. Is failure status guaranteed immediately? Trace the restart-policy conversion needed and identify the waiting window.
4. The first rollback response disappears. Contrast retrying against the same live server with retrying after a fresh default server process. Which evidence would establish operation identity?

Worked answers provide calculations and source-based reasoning. Claims and exact listings are recorded in the companion evidence ledger; their inspected status must not be upgraded without an actual execution record.

Development history and source references

Original development prompts 14, 15, 42 supply the requested scope. The inspected implementation history includes de5900bced92, 430a061fe7f0, 5bfb4ac394a6. Their exported changed paths were compared with this chapter's implementation. These are topical associations; early prompts were retrospective and later work can share a commit. A tracker entry is not evidence that the described experiment ran.

- internal/rollout/rollout.go
- internal/rollout/rollout_test.go
- internal/controlplane/rollback_test.go
- internal/controlplane/operation_lock_test.go

- docs/ROLLOUTS.md

Chapter 18 – Safe Deletion and Resource Cleanup

An operator deletes Atlas API after its rollout experiment. One worker stops its container, then fails during removal. The HTTP delete request has already succeeded, so a second operator assumes the host is clean and reuses a resource. That assumption crosses an acknowledgement boundary: the server accepted deletion intent, but Docker has not confirmed the final side effect. The remaining object is exactly what the deletion workflow must preserve and revisit.

You will follow active, deleting and deleted services; distinguish desired task state from observed removal; identify which process owns retries; and decide what evidence permits finalization. Chapter 12 established Docker identity and Chapter 17 established operation precedence. Here those definitions become a cleanup protocol. The objective is precise convergence without removing unrelated objects, rather than making an API list look empty as quickly as possible.

Soft Deletion Is a State Machine

DeleteService first finds the service in the selected namespace. A missing or invisible service returns not found. An existing deleting or deleted service returns success again. Thus idempotency means repeating the same owned operation does not create a competing cleanup workflow; it does not mean every arbitrary unknown identifier is a successful deletion.

For an active service, the method validates the transition, cancels active deployments, records deleting, and changes tasks toward stopped unless removal is already desired or observed. The deletion-started event records this request-side milestone. The service remains in the memory map; hard deletion is not the mechanism. This retained identity lets later agent reports be correlated with the same service instead of becoming unowned terminal messages.

Unchanged source excerpt — internal/controlplane/controlplane.go:1454–1463:

```
        s.cancelActiveDeploymentsLocked(id, now, "service deletion
requested")
        service.Status = types.ServiceDeleting
        service.UpdatedAt = now
        s.services[id] = service
        for taskID, task := range s.tasks {
                if task.ServiceID == id {
                        if task.DesiredStatus != types.TaskRemoved
&& task.ActualStatus != types.TaskRemoved {
                                if err :=
types.ValidateTaskDesiredTransition(task.DesiredStatus,
types.TaskStopped); err != nil {
                                        return fmt.Errorf("%w: %v",
store.ErrInvalidState, err)
                                }
```

Notice the ordering: deletion wins over the old rollout, then task directives change. The source does not call Docker from inside the server mutex. Doing so would conflate authoritative intent with a remote runtime effect and make a slow daemon block unrelated state operations. Instead the agent retrieves the changed assignments and performs the stop/remove sequence. That design creates a visible waiting state whenever the worker cannot communicate.

The standalone reconciler also has a deletion path and tests. Those are useful package evidence, but default startup does not instantiate that reconciler. Trace MemoryService.DeleteService, ListAssignedTasks, the active agent and ReportTaskStatus for the default binary. A diagram showing the standalone package between every request and agent would teach the wrong ownership graph even if the diagram looked plausible.

Stop and Remove Ownership

The agent's ensureTask checks desired state before attempting normal execution. Stopped or removed intent goes to removeAssignedTask, which resolves a container it is authorized to remove. A provided ContainerID alone is insufficient: inspection must establish that the

managed object's labels match the assigned node and task. If that inspection fails, the runner searches managed containers using the node/task labels and evaluates ownership again.

This guard protects a concrete failure case. A stale or corrupted task report could point at an unrelated container ID. Blindly deleting that ID would make control-plane metadata an unrestricted host deletion command. TestReconcileDoesNotRemoveNonManagedAssignedContainerID exercises the opposite contract: the agent must preserve an object that lacks the required management identity even when the assignment references it.

When a matching container exists, removeLocalContainer stops it and then removes it. An error in either action returns before the agent reports removed. When no matching object exists, the task can still be reported removed, because desired cleanup has no remaining owned object to delete. The distinction is between an object proven absent and an ownership check that forbids deleting an unrelated object; capture drift evidence where those paths differ.

Unchanged source excerpt — internal/agent/agent.go:634–639:

```
func (r *Runner) removeLocalContainer(ctx context.Context,
containerID orchdocker.ContainerID) error {
        if err := r.runtime.StopContainer(ctx, containerID,
10*time.Second); err != nil {
                return err
        }
        return r.runtime.RemoveContainer(ctx, containerID, true)
}
```

The Engine adapter tolerates already-stopped and already-missing objects. That is necessary for retry after a lost response: Docker might have completed an action even though the agent saw an error. A repeated stop/remove must not require resurrecting the object or treating absence as a new failure. It also must not broaden its selector from one task to every managed container on the host.

Retries After Partial Cleanup

Return to the failed removal. At the first poll, the container is running and desired state is stopped. Stop succeeds; remove fails. The agent returns an error and does not send the final removed report. Server state therefore remains deleting with an unfinished task, while Docker now holds a stopped object. Those observations disagree in a useful way: the workflow has progressed, but final acknowledgement is intentionally withheld.

At the next successful poll, identity still points to the same task. Inspecting the stopped object is sufficient to locate it; stop is harmless on repetition, removal can succeed, and only then is removed reported. If removal succeeded but the report response disappeared, the next poll can find no object and repeat the removed report. These are two different retry windows with the same convergence goal.

Interruption	Server evidence	Docker evidence	Required next observation
Before agent poll	deleting, desired stopped	May still run	Assignment retrieval and owned stop
After stop, before removal	Task not removed	Stopped object remains	Retry exact object removal
After removal, before report	Task not removed	Object absent	Repeat scoped removed report
After accepted report	Finalization can advance	Object should remain absent	Verify all service tasks, not just one

A network partition delays this sequence. Neither an accepted delete nor a central timeout remotely fences the container. Operators must avoid interpreting “deleting” as a guarantee that the old process cannot serve or write. For a resource that must not have concurrent writers, observed host cleanup or a separate fencing mechanism is required before ownership is reused.

The runner returns the first task execution error after handling assigned tasks and cleanup. The surrounding loop retries through later reconciliation, rather than repairing the server's records by pretending success. This is an important reason to preserve errors and task identity in the incident worksheet. Suppressing the error would make a lingering object indistinguishable from successful cleanup.

When Finalization Is Safe

MemoryService.maybeFinalizeServiceDeletionLocked requires every task belonging to the service to have ActualStatus removed. Stopped and failed are insufficient in this default path. If the service has no tasks, the loop has no blocker and deletion can complete. The condition is explicit and stronger than a prose description that says “terminal or removed.” Use the actual implementation's distinction.

Unchanged source excerpt — internal/controlplane/controlplane.go:2646–2658:

```
        service, ok := s.services[serviceID]
        if !ok || service.Status != types.ServiceDeleting {
                return
        }
        for _, task := range s.tasks {
                if task.ServiceID != serviceID {
                        continue
                }
                if task.ActualStatus != types.TaskRemoved {
                        return
                }
        }
        if err := types.ValidateServiceTransition(service.Status,
types.ServiceDeleted); err != nil {
```

ReportTaskStatus invokes finalization after accepting removed, and detaches task volumes at that point. Its task/node ownership and lifecycle guards still apply. A late running report cannot resurrect stopped ownership merely because Docker once had a live object. Tests cover

failed-task removal during deletion and rejection of a running resurrection, making the permitted cleanup transition visible without redefining the lifecycle.

The service is then marked deleted and an event is appended; the retained record is not necessarily erased immediately. Retention is a later concern with its own preservation rules. Container deletion also does not automatically imply destruction of all volume data. Local-volume claims, attachment bookkeeping and actual named-volume data have separate lifetimes; Chapter 32 examines that distinction. An operator should never use broad volume pruning as the hidden final step of service deletion.

A fresh default server process complicates the protocol because it loses the service and task maps. Surviving managed Docker objects can outlive the records that explain why they should be removed. This is a process-durability limitation rather than evidence that labels are useless. Labels support adoption and scoped cleanup while identities remain available; they do not reconstruct every desired replica count, deployment history or secret reference.

A cleanup audit should also distinguish request identity from object ownership. Namespace selection restricts which service the caller sees, while the worker label check restricts which runtime object the agent may remove. Those protections occur at different layers and neither replaces the other. An administrator reading an old incident record should match the original namespace, service ID, task ID, node ID and full container ID before taking a manual action. Reusing a friendly service name after a previous run is insufficient, because a new service can have the same name and a different owner identity. Retaining the deletion record makes that comparison possible until retention deliberately expires it.

Atlas Lab: Preserve an Unrelated Container

Start with the disposable pinned source copy and Go 1.25. The smallest useful experiment runs the current agent, Docker-adapter and control-plane fixtures. Keep an explicit list of the named tests rather than interpreting an unrelated green suite as evidence of this failure window.

```
go test ./internal/agent -run
'TestReconcile(RemovesAssignedStoppedTask|
DoesNotRemoveNonManagedAssignedContainerID|
RemovesUnassignedManagedContainers|IgnoresNonManagedContainers)$' -
count=1 -v
go test ./internal/docker -run
'Test(StopContainerAlreadyStoppedIsSuccess|
RemoveContainerAlreadyRemovedIsSuccess)$' -count=1 -v
go test ./internal/controlplane -run 'Test(DeleteService|
AgentTaskStatusAllowsFailedTaskRemovalDuringDeletion|
AgentTaskStatusCannotResurrectStoppedTask)' -count=1 -v
```

Expected assertions establish idempotent adapter behavior, label-aware agent removal, repeated deletion intent and finalization after agent return. The fresh package-test record records the fake/local assertions with caching disabled and explicitly lists the unconfigured real-Docker integration test as skipped. They do not by themselves inject a real daemon failure. A book-side failure fixture may wrap the fake runtime to fail the first RemoveContainer and delegate the second call; it is a teaching extension and needs its own acceptance test. Do not edit the immutable application fixture or present an unimplemented wrapper as an observed result.

For the separate real Atlas experiment, create only run-owned containers and a deliberately unrelated sentinel. Record each full ID before requesting deletion. Pause only the owned agent to observe the deleting window, resume it, then inspect the specific task containers and sentinel independently. If using a disposable daemon or proxy to inject removal failure, document that boundary and verify the failure happened at remove rather than at stop or assignment retrieval.

Abort when a selector contains no recorded task/node identity, an inspected object lacks expected labels, or the sentinel changes. Cleanup must remove the exact run-owned residual IDs after restoring connectivity. There is no legitimate broad-prune shortcut for a missing acknowledgement. Store request/status snapshots, runtime inspection and final sentinel proof separately, with UTC timestamps and command exits. Until such a record is attached, these remain expected experiment instructions.

Exercises

1. Why does repeated delete of an existing deleting service succeed while delete of an unknown service may return not found?
2. A stopped container remains after the first remove failure. Identify which component retries and why the server must not finalize yet.
3. A task's stored ContainerID points at an unmanaged sentinel. Explain why ownership labels are checked before removal and what evidence should remain.
4. After all task records say stopped, an operator deletes every local Docker volume. Identify both unsupported assumptions and design a scoped finalization observation instead.

Worked answers justify the cleanup decisions. The chapter's evidence ledger remains source-inspected until the specific failure injection and runtime observations have matching command records.

Development history and source references

Original development prompts 16, 25 supply the requested scope. The inspected implementation history includes 2084f5f77bd3, 66cfb7049fc1. Their exported changed paths were compared with this chapter's implementation. These are topical associations; early prompts were retrospective and later work can share a commit. A tracker entry is not evidence that the described experiment ran.

- internal/reconciler/reconciler.go

- internal/controlplane/controlplane.go
- internal/agent/agent.go
- internal/docker/docker.go
- docs/RECONCILER.md
- docs/RELIABILITY.md

Chapter 19 – Offline Nodes, Draining, and Rescheduling

Atlas needs a worker upgrade. During planned maintenance the operator can ask the control plane to drain that worker and observe replacements. During an unexpected partition, the server instead sees silence and must infer whether to replace work. These situations may produce similar replica movement, but they carry different knowledge. A cooperative worker can acknowledge stopping; an isolated worker may continue running while the server declares it offline.

This chapter follows the active node monitor, drain progress and late-container cleanup. You will calculate the timeout boundary, distinguish drain from disappearance, and explain why replacement is not fencing. Prerequisites are the scheduler and agent chapters plus Chapter 18's separation of deletion intent from actual removal. Atlas remains stateless for replacement experiments; stateful ownership is a deliberate boundary, not a test to perform on valuable data.

Heartbeat Expiry and Offline Status

Default orch-server constructs node.NewMonitor with configured HeartbeatTimeout and NodeMonitorInterval and runs it in a goroutine. The architectural roadmap contains older wording treating monitoring as future work; current startup and internal/node/monitor.go are the authority. Package presence would not be sufficient evidence, but here the executable caller is explicit.

Monitor.Run calls CheckOnce immediately, then checks on its ticker. An initial check error returns from the loop; later pass errors are logged and the loop continues. Server startup logs a stopped monitor rather than silently supervising a restart. Operators should therefore distinguish a monitor that is healthy but has no stale nodes from one that stopped before entering its recurring checks.

MemoryService.MarkStaleNodesOffline computes cutoff as now minus timeout. It examines ready or draining nodes and skips a zero heartbeat. A heartbeat must be strictly before cutoff; equality is not stale in this comparison. With a thirty-second timeout and a heartbeat at noon, a check exactly at 12:00:30 does not mark the node offline. A later check can. The five-second default monitor cadence adds observation phase, so thirty seconds is not an exact externally guaranteed detection latency.

Unchanged source excerpt — internal/controlplane/controlplane.go:681–690:

```
        cutoff := now.Add(-timeout)
        offline := make([]types.Node, 0)
        for id, node := range s.nodes {
                if node.Status != types.NodeReady && node.Status !=
types.NodeDraining {
                        continue
                }
                if node.LastHeartbeatAt.IsZero() || !
node.LastHeartbeatAt.Before(cutoff) {
                        continue
                }
                if err := types.ValidateNodeTransition(node.Status,
types.NodeOffline); err != nil {
```

After a valid transition, the method records offline, updates the node, marks its active tasks lost, emits node.offline.detected and reconciles services. Repeating the method does not repeatedly transition an already-offline node through the same branch. This guards one state mutation but does not prevent every source of duplicate external effects; runtime acknowledgement is elsewhere.

Heartbeat age is server-side evidence about the last accepted report. A metrics endpoint responding is different evidence; an unreachable metrics target is not the same predicate as stale LastHeartbeatAt. Collect both when troubleshooting, and record the configured interval/timeout rather than relying on remembered defaults after configuration changes.

Drain Versus Node Disappearance

DrainNode is a planned API operation with maintenance and identity checks. It marks the node draining and immediately reconciles replacement work. Ready-node selection excludes the draining worker. If there are no ready replacements, the workflow stays pending and emits a warning rather than pretending a task has safely moved.

The memory drain path counts off-node active tasks for each service, creating assigned replacements on ready nodes until the desired count is represented. It then waits for available off-node tasks before requesting stops on the old worker. In this implementation available means running or healthy; the drain test explicitly advances a replacement with a running report. Do not transfer Chapter 17's new-rollout-target healthy-only predicate to this workflow.

Unchanged source excerpt — internal/controlplane/controlplane.go:2401–2415:

```
                if availableTaskCount(offNodeActive) <
service.Spec.Replicas {
                        continue
                }
                for _, task := range drainingTasks {
                        if task.DesiredStatus == types.TaskStopped
|| task.DesiredStatus == types.TaskRemoved {
                                continue
                        }
                        if err :=
types.ValidateTaskDesiredTransition(task.DesiredStatus,
types.TaskStopped); err != nil {
                                continue
                        }
                        task.DesiredStatus = types.TaskStopped
                        task.UpdatedAt = timestamp
                        s.tasks[task.ID] = task
                }
        }
```

TestDrainCreatesReplacementAndStopsOldTaskAfterReplacementRuns is precise about the ordering: the old task starts on node A; node B is registered; drain creates a distinct replacement; reporting that

replacement running changes the old desired status to stopped. The task ID is new even when service identity is stable. The test proves memory-state ordering, not packet-level availability or a physical worker shutdown.

Drain progress also needs careful interpretation. nodeDrainStatusLocked counts RemainingTasks whose desired status is neither stopped nor removed. Once those directives are set, it can report DrainComplete even while the old Docker object has not yet acknowledged removal. That result is completion of the control-plane drain decisions, not proof that a technician can instantly unplug the host without checking runtime consequences.

Uncordon sets the node ready again. The fixture that uncordons before replacement readiness confirms the old task can remain desired running and drain status becomes not_draining. This is a useful cancellation window, but it is not a transactional rollback of every replacement ever created. Inspect both cohorts before interpreting uncordon as restoration of the original layout.

Replacement Tasks and Rescheduling

When a heartbeat timeout marks tasks lost, each active task gets a node_lost condition. For a stateless service, the memory method sets desired removed, actual failed, failure reason node_lost and a finish timestamp. Reconciliation can then create replacement identity on a ready node. The original task record remains the reference for late-report guards and returning-worker cleanup.

Unchanged source excerpt — internal/controlplane/controlplane.go:2078–2092:

```
                task.Conditions =
upsertTaskCondition(task.Conditions, types.TaskCondition{
                        Type:
types.TaskConditionNodeLost,
                        Message:            "node heartbeat timed
out",
                        LastTransitionTime: timestamp,
                })
                service := s.services[task.ServiceID]
                if !service.Spec.Stateful {
                        task.DesiredStatus = types.TaskRemoved
                        task.ActualStatus = types.TaskFailed
                        task.FailureReason = "node_lost"
                        task.FinishedAt = timestamp
                }
                task.UpdatedAt = timestamp
                s.tasks[taskID] = task
        }
```

For Stateful services, that mutation is intentionally withheld: the condition and updated timestamp remain, but the old task is not automatically converted to removed/failed by this branch. TestMarkStaleNodesOfflineDoesNotReplaceStatefulTask captures this conservative distinction. The flag does not implement distributed storage, leader fencing or data recovery. It simply avoids one automatic replacement policy that would otherwise risk competing writers.

Normal service replacement still uses MemoryService's immediate placement, not the standalone planner's full resource-fit logic. “Insufficient capacity” in the drain progress response principally checks whether ready replacement nodes exist in this path; it must not be read as proof that CPU, memory, host ports and local data have all been globally reserved. Trace the exact fields when deciding whether a proposed maintenance move can fit.

On return, a worker can heartbeat back to ready. Its agent compares local managed objects with current assignments and removes objects no longer assigned to its node.

TestReturnedNodeRemovesNodeLostUnassignedContainer exercises that later cleanup. The original and replacement task IDs must remain distinct in the worksheet, otherwise an operator can confuse adoption of a still-owned object with removal of obsolete work.

Partitions Without a Fencing Promise

Consider two workers, A and B. Atlas serves on A. A loses connectivity to the server, but its Docker daemon and application keep running. After the heartbeat deadline, the server marks A offline and replaces stateless work on B. B starts a new task. Both HTTP endpoints can now exist even though the server's current desired set excludes A's task.

There is no contradiction: central records and remote side effects have different owners. A failed or removed record is not a remote power switch. Without connectivity, A cannot necessarily receive the directive or report its cleanup. After the partition heals, label-aware reconciliation can remove the obsolete container. It cannot undo requests or writes processed during overlap.

For a stateless HTTP service, temporary overlap may be acceptable if clients and data stores tolerate it. For a single-writer workload, it can violate a safety invariant. A real fencing design might use storage generations or an external authority that rejects stale writes, but none is implied by the node timeout. Such a design belongs to a separately tested extension, not an optimistic explanation of the baseline.

Situation	Server knows	Server cannot conclude
Planned drain with replacement report	Replacement reached running/ healthy; stop intent can be set	Old runtime already removed
Heartbeat timeout	Last accepted heartbeat predates cutoff	Worker is powered off
Returning agent cleanup	Current assignment set excludes obsolete task	No overlap occurred during partition

Situation	Server knows	Server cannot conclude
Stateful node_lost condition	Owner became unreachable	Data safely moved to another node

An incident narrative should preserve the interval between replacement start and old-container stop. Measuring that interval in an actual lab is useful; inventing a zero-overlap result from unit tests is not. Chapter 34 later separates leader coordination from worker fencing for the same reason.

The late-report boundary deserves its own check. A returning agent credential establishes who sent a message, while TaskID and NodeID establish which assignment it concerns. Lifecycle guards then decide whether the requested status can advance that particular record. Authentication alone cannot make an obsolete running report authoritative. Conversely, an idempotent report for still-owned work may legitimately be accepted. Test the desired/actual state combination produced by your fault instead of claiming that every report older than a timeout is rejected. The relevant question is current ownership and permitted transition, not whether the packet feels stale to an observer.

Atlas Lab: Pause One Owned Agent

Use Go 1.25 and the disposable pinned source copy for deterministic fixtures. No physical worker is needed for the first stage. Source-checked tests expose stale detection, stateless/stateful policy, pending drain and returning cleanup.

```
go test ./internal/node -run TestMonitor -count=1 -v
go test ./internal/controlplane -run 'Test(MarkStaleNodesOffline|
HeartbeatReturnsOfflineNodeReady|Drain|
UncordonBeforeDrainCompletion|NodeOfflineDuringDrainStatus)' -
count=1 -v
go test ./internal/agent -run
TestReturnedNodeRemovesNodeLostUnassignedContainer -count=1 -v
```

The fresh package-test record includes these node/control-plane cases with caching disabled. Their fake clocks and records establish the selected transition scenarios; they do not pause an actual worker or fence a remote process. For an actual two-agent Atlas run, record server/agent PIDs, node IDs, task IDs, direct endpoints and Docker IDs before the fault. Pause only the recorded disposable agent process; never stop a production host, shared daemon or host firewall. Keep a bounded pause and an explicit resume/terminate cleanup handler.

During the pause, sample LastHeartbeatAt, node status, lost-task condition, replacement identity and direct reachability of the old endpoint. Do not assume a particular detection timestamp; retain actual samples. Resume the agent, observe the new assignment set and check the old container's removal independently. Abort if replacement capacity is absent, the selected process is not run-owned, or any operation targets a stateful user workload.

The planned-maintenance variant drains A while A remains connected. Record replacement readiness and old desired stop, then inspect physical cleanup. Uncordon only the recorded node if the test requires cancellation. At the end, resume every paused process, stop the run-owned agents and remove exact lab objects. A trustworthy result explains residual effects rather than hiding them behind a final green status.

Exercises

1. Why does a heartbeat exactly at the timeout cutoff remain eligible on that check, and how does the monitor cadence affect observed detection?
2. A drain reports complete while Docker still lists the old container. Is that necessarily a bug? Name the completion predicate and the missing observation.
3. Explain the stateless and Stateful mutations after node loss. Which one proves exclusive access to data?

4. Design a partition worksheet that can demonstrate overlap without touching external hosts. Which identities and two independent endpoint observations must it retain?

Worked answers connect these decisions to the exact guards. The source evidence supports a timeout-and-reconciliation mechanism with explicit limits, not an invisible worker-fencing protocol.

Development history and source references

Original development prompts 39, 40 supply the requested scope. The inspected implementation history includes 42d4caa4451f, a5d4bfb37f70. Their exported changed paths were compared with this chapter's implementation. These are topical associations; early prompts were retrospective and later work can share a commit. A tracker entry is not evidence that the described experiment ran.

- internal/node/monitor.go
- internal/node/monitor_test.go
- internal/controlplane/drain_test.go
- internal/controlplane/offline_test.go
- internal/controlplane/operations.go
- docs/OPERATIONS.md

Chapter 20 – Endpoint Discovery and Internal DNS

After the worker experiment, Atlas API still has a stable service name but different task locations. A client that retained yesterday's node address cannot reach today's replacement merely because the control plane reports the desired replica count. Discovery must expose the current endpoint set, and the client must interpret that set correctly. Adding a DNS name improves naming, but it does not create a virtual network or eliminate published-port identity.

You will derive endpoints from task reports, compare the HTTP discovery builder with the optional DNS implementation, and analyze a stale-cache window after rescheduling. Chapters 14 and 19 supply health and node-loss vocabulary. The central question is which observed fields make an address useful, and which facts the response does not establish. This is the current host-port model, with no implied overlay or Kubernetes-compatible DNS service.

Endpoint Sets and Health Filtering

BuildServiceEndpoints receives one service, tasks and nodes. It indexes nodes by ID, then ignores tasks for another service or tasks without a node assignment. Its includeTask helper requires desired running before interpreting actual status. Running and healthy are included by default; unhealthy is included only when the caller explicitly requests it. Pending, failed, stopped and removed are not useful service endpoints here.

Unchanged source excerpt — internal/discovery/discovery.go:98–110:

```
func includeTask(task types.Task, includeUnhealthy bool) bool {
        if task.DesiredStatus != types.TaskRunning {
                return false
        }
        switch task.ActualStatus {
        case types.TaskRunning, types.TaskHealthy:
                return true
        case types.TaskUnhealthy:
                return includeUnhealthy
        default:
                return false
        }
}
```

For each accepted task the builder finds its node and iterates task ports. It prefers task-specific ports when present, falling back to service-spec ports otherwise. That distinction matters when dynamic host ports were assigned: the workload's container port can remain80 while different tasks publish on different host ports. An endpoint response must describe the actual task assignment rather than repeat a public-zero request from the service specification.

The emitted record carries service name/ID, task ID, node ID/address, public host port, container port, protocol, health status and service version. Sort order is deterministic by service name, task, container port, protocol and published port. Stable ordering is useful for diffing; it is not sticky load balancing or a guarantee that two sequential responses represent one atomic runtime snapshot.

Node presence is required, but this builder does not independently exclude a node merely because Node.Status is offline. The ordinary lost-node path changes task state, which usually removes it through the task predicate. Those are different claims. If a fixture deliberately supplies an offline node with a running desired-running task, inspect the builder's result instead of assuming an unstated node-health filter.

BuildAllServiceEndpoints additionally skips services with a nonempty non-active status. It sorts the services it receives. Developers embedding the helper should notice that it sorts that slice, whereas other helpers copy before sorting. The practical contract is about both visible records and inputs; do not add a concurrent caller that assumes the slice is untouched without testing ownership.

Namespace Names and DNS Records

The optional internal DNS server is activated by ORCH_DNS_ADDR in default main. TTL defaults to thirty seconds, with a positive configured duration parsed by startup. The resolver supports service.svc.orch in the default namespace and service.namespace.svc.orch for an explicit namespace. It lowercases and trims a trailing dot. Other name shapes are rejected rather than interpreted as general Kubernetes search-domain behavior.

DNS Resolve selects the namespace context and finds an active service by its name. It lists that service's tasks and the available nodes, then extracts node advertise addresses from running or healthy tasks. This is a separate implementation, not a call to discovery.BuildServiceEndpoints. In particular, Resolve checks actual running/healthy but does not perform the endpoint helper's desired-running predicate. A task awaiting stop can therefore require separate analysis in the DNS path.

Unchanged source excerpt — internal/dns/server.go:97–121:

```
        out := []net.IP{}
        seen := map[string]struct{}{}
        for _, t := range tasks {
                if t.ActualStatus != types.TaskHealthy &&
t.ActualStatus != types.TaskRunning {
                        continue
                }
                host, _, err :=
net.SplitHostPort(addresses[t.NodeID])
                if err != nil {
                        host = addresses[t.NodeID]
                }
                ip := net.ParseIP(host).To4()
                if ip == nil {
                        continue
                }
                if _, ok := seen[ip.String()]; !ok {
                        seen[ip.String()] = struct{}{}
                        out = append(out, ip)
                }
        }
        return out, nil
}
func parseName(name, defaultNS string) (string, string, bool) {
        parts :=
strings.Split(strings.TrimSuffix(strings.ToLower(strings.TrimSpace(n
ame)), "."), ".")
        if len(parts) == 4 && parts[1] == "svc" && parts[2] ==
"orch" {
                return "", "", false
```

The address conversion splits an advertise host/port where possible, parses the host as an IP and calls To4. Hostnames and IPv6-only addresses are skipped by this implementation. Multiple tasks on the same node contribute one deduplicated IPv4 address, not one DNS answer per replica. A client counting answers to infer replica count will therefore get the wrong abstraction.

ServePacket implements a small DNS response path: it expects one question, decodes labels and emits A answers with the configured TTL. Non-A questions receive no A records; this is not proof of a complete

recursive resolver, SRV discovery or IPv6 service addressing. The server's query/error counters measure this component's work, not application request success.

Node Addresses Still Need Ports

Suppose Atlas task A publishes container 80 as host 31081 on node 192.0.2.10, while task B publishes the same container port as 31082 on 192.0.2.11. These documentation addresses are illustrative, not reachable lab targets. The HTTP endpoint records preserve both pairs. DNS can return the node IPs, but an A record carries no task's host port. Resolving the service name and blindly connecting to port 80 can therefore fail despite a perfectly valid answer.

If every selected node uses one known fixed published port, a client can combine the DNS name with that configured port. With dynamic per-task ports, it needs the richer endpoint API or a gateway that consumes it. Naming and routing are separate mechanisms. A stable logical name does not guarantee a stable socket destination, and a successful lookup does not prove that the firewall, daemon binding or application permits the connection.

The upstream DNS compose example demonstrates a client configured to query the server's DNS address on an isolated compose network. It does not create a service VIP. It also does not supply a complete Atlas deployment by itself; the service, tasks and node advertisements still have to exist in the control plane. Treat the example as topology/configuration evidence and verify all prerequisites before calling it a working application network.

Surface	Address identity	Port information	Important filter
HTTP endpoints	Per task/node record	Public and container ports	Desired running plus actual running/healthy; optional unhealthy
DNS A answers	Deduplicated node IPv4	None	Actual running/healthy in separate resolver

Surface	Address identity	Port information	Important filter
Direct application socket	Specific host and published port	Required	Actual reachability tested independently

These distinctions also explain why include_unhealthy is a diagnostic option rather than a request to route normal traffic indiscriminately. It expands the observed set; it does not fix an unhealthy application or make an offline worker reachable.

Rescheduling, TTL, and Stale Clients

Consider a client caching node A at time zero with a thirty-second TTL. At second ten, the server replaces Atlas on node B and task A becomes obsolete. A new query may already return B, but the earlier client can retain A until its cache expires. Even that expiry does not guarantee an open connection moves: connection pooling, application-level retries and each client's resolver behavior are separate layers.

This timeline is hypothetical arithmetic, not a measured thirty-second recovery. Real effective staleness includes the task report delay, node-monitor cadence, DNS query time, intermediary caches and the client's policy. The TTL field is one bound offered to compliant caches, not a universal reconnection deadline. Lowering it trades query volume for freshness and does not repair a wrong node address or missing port.

A further ambiguity occurs while desired stop and actual running disagree. HTTP discovery can exclude the task through desired status while DNS Resolve still accepts its actual status. That divergence is source-derived and testable with a synthetic task. Do not silently claim both surfaces use identical filters just because documentation groups them under service discovery. A review can propose sharing the helper later, but that would be a change outside this pinned baseline.

For an operator, a useful incident record saves three snapshots: the control-plane endpoint list, the DNS answer with TTL and the direct socket outcome for the exact returned host/port. Keep task and service versions alongside them. A lookup failure, a stale answer and a refused application connection suggest different recovery owners and should not collapse into one message saying “DNS is broken.”

Namespace selection also differs from authorization. Encoding a namespace in the DNS name selects which service records to inspect; it does not present a user JWT or prove that the querying client has an API role. Bind the experimental resolver only where intended clients can reach it. Do not describe the namespace spelling itself as confidential tenant isolation.

Atlas Lab: Compare Two Selection Functions

Use Go 1.25 and the disposable pinned source copy. The deterministic phase needs no real Docker or external DNS. Inspect the fixtures that build healthy/unhealthy tasks and verify task-specific assigned ports. The fresh package-test record includes discovery and DNS cases with caching disabled. It records package assertions over fixture data, separately from the optional owned-listener experiment below.

```
go test ./internal/discovery -run
'TestBuildServiceEndpoints(FiltersByHealth|UsesTaskAssignedPorts)$'
-count=1 -v
go test ./internal/dns -run
'TestResolve(HealthyEndpointsAndDNSPacket|RejectsUnsupportedName)$'
-count=1 -v
```

Add an explicitly separate teaching fixture for desired-stopped/actual-running if you want to demonstrate the filter divergence. It should feed the same synthetic identities into each function and assert the source-derived difference, with no network dependency. Keep that fixture outside immutable evidence and distinguish its result from the existing upstream tests.

For a live owned Atlas environment, first save service/task/node records and exact published ports. Query the loopback API's endpoint surface, then query only the owned DNS listener. Configure a free nonprivileged local UDP port rather than assuming permission to occupy host port 53. Compare answers before and after a run-owned task's health transition. Request a direct application endpoint separately using the recorded host/port; never send test traffic to illustrative documentation addresses.

Abort if advertisements resolve to an unowned host, the DNS listener conflicts with an existing service, or the client configuration would change the host's global resolver. Cleanup closes the owned listener and restores only the disposable client configuration. Save answer bytes or parsed records, TTL, UTC times and command exits. Captured DNS names improve observability, but the experiment cannot establish an overlay, service VIP, universal cache behavior or kernel network isolation.

Exercises

1. Two healthy Atlas tasks share a node but use different public ports. How many node-IP answers can DNS return, and why is that insufficient endpoint identity?
2. Predict HTTP discovery and DNS selection for a task with desired stopped and actual running. Which implementation difference controls the result?
3. An advertise address is an IPv6 literal or hostname. Explain what this DNS resolver does and why API endpoint presence does not contradict it.
4. A client still reaches the old worker after the new endpoint appears. Design evidence that distinguishes TTL caching, pooled connections and a wrong public port without claiming a universal recovery deadline.

Worked answers retain the different filtering and addressing boundaries. Future networking designs can improve them, but the current book must describe the socket that actually exists.

Development history and source references

Original development prompts 35, 78 supply the requested scope. The inspected implementation history includes 03f8c30f6df8, 4d641bbe8913. Their exported changed paths were compared with this chapter's implementation. These are topical associations; early prompts were retrospective and later work can share a commit. A tracker entry is not evidence that the described experiment ran.

- internal/discovery/discovery.go
- internal/discovery/discovery_test.go
- internal/dns/server.go
- internal/dns/server_test.go
- docs/SERVICE_DISCOVERY.md
- deployments/examples/internal-dns-compose.yaml

Chapter 21 – Ingress, Host Networking, and the Overlay Roadmap

Atlas now has discoverable task endpoints, but visitors should use an application host name rather than track worker addresses and dynamic ports. The optional Traefik integration supplies that gateway. Its usefulness depends on understanding the actual packet path: a client reaches Traefik, Traefik selects a node-level published port, and Docker forwards to the container. A route definition is neither an overlay network nor a guarantee that every hop is reachable.

This chapter builds on endpoint identity from Chapter 20 and security-policy scope from Chapter 26 when following the operator/architect reading path. You will derive the generated router and service fields, identify failures at gateway and worker boundaries, and separate the executable host-port model from proposed networking improvements. Atlas remains the same stateless API; we change how clients find it, not the orchestrator's underlying runtime architecture.

The Current Host-Port Packet Path

Docker's adapter maps container ports to published host ports in HostConfig.PortBindings. Ordinary exposure does not require host network mode: Docker port publication and explicitly selecting host networking are different configuration paths. The latter is a privileged policy choice considered in the admission chapter. Calling the whole current model “host-port networking” should not erase that distinction.

A routed Atlas task has a container port describing where the application listens and a public host port describing where a gateway can reach it. The node advertise address may include the agent's own port. That agent port is not the application backend destination; the Traefik builder extracts the host and combines it with the endpoint's PublicHostPort.

Unchanged source excerpt — internal/traefik/config.go:143–156:

```
func endpointURL(endpoint discovery.Endpoint) (string, bool) {
        if endpoint.PublicHostPort <= 0 {
                return "", false
        }
        nodeAddress := strings.TrimSpace(endpoint.NodeAddress)
        if nodeAddress == "" {
                return "", false
        }
        host := addressHost(nodeAddress)
        if host == "" {
                return "", false
        }
        return "http://" + net.JoinHostPort(host,
strconv.Itoa(endpoint.PublicHostPort)), true
}
```

This transformation is why a node advertised as http://192.0.2.10:8081 can produce a backend on another port. The documentation address is illustrative; it must never be used as a real experiment target. The builder emits an HTTP URL regardless of the route's client-facing TLS flag. That flag controls the router's TLS configuration, not an automatic HTTPS upgrade for the worker backend.

A complete path has four separately observable boundaries: client to gateway listener, gateway route match, gateway to published worker port and Docker to application listener. Test each one with the exact owned address and Host header. An API response that lists a healthy task cannot establish that firewall rules permit the gateway's connection, or that the application accepts the requested path.

Traefik Routers and Services

BuildConfig first indexes endpoint sets by ServiceID and sorts a copy of services. It ignores non-active services, validates each route and gathers matching servers. A route without usable servers is skipped, so the output can omit both its router and service. This is preferable to

imagining an empty configuration entry is a functioning route, but clients will still observe an unavailable application until a usable backend appears.

Each router contains a Rule and Service name, with an optional empty TLS configuration object. routeRule combines Host and PathPrefix using quoted values. The related service contains loadBalancer.servers and passHostHeader true. Server URLs are sorted, providing deterministic configuration output. These are concrete Traefik provider fields, not an implementation of a custom reverse proxy inside orch.

Unchanged source excerpt — internal/traefik/config.go:71–85:

```
serviceName := routerName + "-service"
router := Router{
        Rule:    routeRule(route),
        Service: serviceName,
}
if route.TLS {
        router.TLS = &TLSConfig{}
}
config.HTTP.Routers[routerName] = router
config.HTTP.Services[serviceName] = Service{
        LoadBalancer: LoadBalancer{
                Servers:        servers,
                PassHostHeader: true,
        },
}
```

The endpoint selector requires matching container port, TCP protocol, actual running or healthy status and a positive published port with usable node host. Upstream discovery ordinarily already filtered desired-running work before the builder sees it. Preserve that caller relationship: BuildConfig's own input contract and the API's assembled endpoint set are different layers of protection.

The fixture TestBuildConfigIncludesOnlyHealthyTCPRouteEndpoints provides a useful adversarial mix: one running target, one healthy target, one unhealthy task, one wrong-port task and one UDP task. Only two HTTP backend URLs survive. The test asserts the exact router rule, a

nonnil TLS object, passHostHeader and sorted URLs. TestBuildConfigSkipsRoutesWithoutHealthyServers asserts no routers/services for an unhealthy-only set.

A generated route key incorporates service name, host, port and route index after slugging. Readers considering broad multitenant compositions should inspect that naming scheme rather than assume it includes every namespace/UUID dimension. The current chapter does not claim arbitrary cross-namespace route-name uniqueness. That is a concrete integration review question, distinct from whether one Atlas route is generated correctly.

Failures at Node and Gateway Boundaries

The provider endpoint is GET /v1/integrations/traefik/config. The upstream compose profile configures Traefik to poll it every two seconds. This creates another freshness boundary: an unhealthy task can disappear from the server's latest configuration before the gateway has fetched and applied that change. A previous gateway configuration can still send requests to a now-bad destination during the interval.

The two-second setting is an example poll cadence, not a universal routing-convergence promise. Gateway processing, transport errors, workload report timing and persistent connections can extend effective behavior. If user JWT authentication is enabled on the provider endpoint, the external consumer also needs a compatible authorized fetch path. The default demonstration's unauthenticated-user configuration cannot be assumed valid after security settings change.

Consider Atlas on node A and node B while the gateway loses access to node A's public port. Both agents may still report healthy probes because probes originate locally on the workers. The gateway receives a route containing A but cannot connect to it. This is a data-plane reachability failure that a local health report does not disprove. Record the failed gateway-to-A connection independently rather than changing desired replicas before understanding the hop.

A different failure occurs when all endpoints are filtered out: the provider output omits the route. Query the provider JSON first, compare current task versions/states and published ports, then inspect gateway behavior. A client using the wrong Host header can fail even when backends are present. That is why an experiment must preserve the request's actual host and path, not merely report that curl returned an error.

Observation	Likely boundary to inspect	What it does not prove
Provider has no route	Service status, route validation, endpoint filtering	Gateway itself is broken
Provider has backend, gateway refuses it	Node host/public port/ firewall/app listener	Agent health report was forged
Direct backend works, host request fails	Router match, provider poll, gateway listener	Overlay networking is required
TLS router object exists	Client-facing TLS intent	Certificate issuance or backend encryption occurred

The last row is especially important. A boolean in a workload route can request TLS configuration, but the builder does not issue certificates, distribute trust or encrypt every internal connection. Operators must know which external gateway configuration owns those responsibilities.

An Overlay or CNI Design Is a Separate Step

The advanced networking document begins from these limitations: scarce public ports, exposed node topology, clients following rescheduling and missing workload network identity/policy. It proposes task addressing, IPAM, overlays such as VXLAN or WireGuard and possible CNI plugin integration. Those are design alternatives, not active branches hidden behind the existing route endpoint.

An overlay would need stable task address allocation, checkpointed lifecycle, route setup/removal, MTU handling and recovery after partitions. CNI integration would add plugin versioning, output validation, deadlines, privilege boundaries and repeatable cleanup. Network policy would

require an enforcement mechanism tied to actual workload identity. Writing a namespace selector in a design does not make packets tenant-isolated in the current Docker host-port model.

These proposals interact with the ownership questions already studied. If a server loses task records, who reclaims an address? If an isolated worker continues serving, which generation may retain a route or tunnel key? If a plugin succeeds but its response is lost, how is cleanup repeated? A useful networking design answers these failure windows rather than only drawing a new subnet.

Atlas does not need an invented overlay to demonstrate ingress. Keep the baseline example small and inspectable, then write a separate architecture decision describing the missing mechanism you would add. A proposed feature gate must not appear in a source-derived command unless the pinned executable actually accepts it. The benefit of source-first study is knowing exactly where an extension begins.

Port pressure should be measured at the node boundary. Two containers can share the same internal application port while requiring distinct published ports on one host. Moving one replica to another node changes that collision domain, but it also changes the address the gateway must use. A capacity plan that counts only CPU and memory misses this constraint. Likewise, adding replicas cannot repair a gateway route whose selected destination port is wrong; more bad endpoints only multiply the same configuration error.

Atlas Lab: Inspect Configuration Before Starting a Gateway

The deterministic stage uses the disposable pinned source copy with Go 1.25. It needs no external gateway or Docker daemon and proves the builder's transformation/selection contract.

```
go test ./internal/traefik -run
'TestBuildConfig(IncludesOnlyHealthyTCPRouteEndpoints|
SkipsRoutesWithoutHealthyServers)$' -count=1 -v
go test ./internal/api -run
TestTraefikConfigUsesHealthyServiceRouteEndpoints -count=1 -v
```

The fresh package-test record includes Traefik configuration tests with caching disabled. Those fixtures exercise generated data rather than a running gateway or encrypted client connection. Inspect the upstream traefik-demo manifest and compose profile, especially the listener mappings, provider URL and poll interval. Adapt only the service name to the owned Atlas case when creating a separately labeled teaching manifest; preserve supported fields and record the change.

For the real gateway variant, use an isolated daemon/network or explicitly owned compose project, verify free listener ports, and record the service/task/container identities. Do not start the upstream profile against an unrelated existing project just because it is convenient. Compare provider JSON, direct backend requests and gateway requests with the configured Host header. Induce a missing-endpoint case with the owned unhealthy workload rather than shutting down the host network.

Abort if any advertised backend is not run-owned, if compose would replace an existing service, or if the selected ports conflict. Cleanup stops only the chosen compose project and recorded Atlas workload objects; preserve unrelated volumes and daemon state. Save configurations and observations before cleanup. A successful gateway request demonstrates that concrete path at that time, not a service VIP, overlay, CNI implementation or comprehensive tenant network policy.

Exercises

1. Why is a node's advertised agent port discarded when constructing an application backend URL?
2. A TLS route emits http:// backend servers. Explain why this is consistent with the builder and what must be configured separately.
3. A task is locally healthy but unreachable from the gateway. Give a hop-by-hop investigation that preserves both observations.
4. Select one proposed overlay/CNI mechanism and define the crash window its acceptance test must cover before it could replace the current host-port behavior.

Worked answers separate generated intent, gateway configuration and actual packet effects. That distinction lets an operator use the current integration while an architect evaluates future networking honestly.

Development history and source references

Original development prompts 36, 77 supply the requested scope. The inspected implementation history includes ea0760e8201c, 4d641bbe8913. Their exported changed paths were compared with this chapter's implementation. These are topical associations; early prompts were retrospective and later work can share a commit. A tracker entry is not evidence that the described experiment ran.

- internal/traefik/config.go
- internal/traefik/config_test.go
- docs/TRAEFIK.md
- docs/NETWORKING.md
- docs/NETWORKING_DESIGN.md
- deployments/examples/routed-api.yaml
- deployments/examples/traefik-demo.yaml

Chapter 22 - Autoscaling Decisions and Missing Integration

Atlas receives a burst of requests. An operator sees CPU utilization rise and expects the autoscaling fields in its specification to add replicas. The source edition contains a meaningful CPU recommendation controller and a fake metrics provider, yet default orch-server never starts that controller. Understanding the equation is valuable; expecting the default binary to execute it is a composition mistake. The first operational diagnosis is therefore whether there is an active consumer of the policy.

You will calculate scale-up and bounded scale-down recommendations, trace cooldown and rollout exclusion, and identify production metrics and wiring gaps. This chapter follows scheduler arithmetic, rollout ownership and quota accounting. It treats autoscaling as an implemented package with tests, not as an automatically enabled production feature. Atlas provides concrete numbers for reasoning without sending artificial load to the user's host.

The CPU Recommendation Equation

MetricsProvider.ServiceCPUUtilization receives a service and returns one floating-point utilization value or an error. The controller does not define that value's measurement window by itself. The fake tests use values 100, 50 and 25 as percentages in the same scale as TargetCPUUtilization; supplying a fraction 0.7 where the target is 70 would produce a different and incorrect interpretation. A real provider must establish units before the controller's arithmetic can be trusted.

recommendation first raises current to MinReplicas when necessary. For a positive target, it computes the ceiling of current multiplied by observed utilization divided by the target. It then clamps to the minimum and, when

positive, the maximum. The function is deterministic over these inputs; it does not consult Docker, node free capacity, request rate or a percentile distribution.

Unchanged source excerpt — internal/autoscaler/autoscaler.go:213–228:

```
func recommendation(current int, cpuUtilization float64, policy
types.AutoscalingPolicy) int {
        if current < policy.MinReplicas {
                current = policy.MinReplicas
        }
        recommended := current
        if policy.TargetCPUUtilization > 0 {
                recommended = int(math.Ceil(float64(current) *
cpuUtilization / float64(policy.TargetCPUUtilization)))
        }
        if recommended < policy.MinReplicas {
                recommended = policy.MinReplicas
        }
        if policy.MaxReplicas > 0 && recommended >
policy.MaxReplicas {
                recommended = policy.MaxReplicas
        }
        return recommended
}
```

Let Atlas have three replicas, target 60 and observed utilization 90. The raw result is ceil(3×90/60), or ceil 4.5, producing five. With maximum four, the final recommendation is four. That clamp prevents desired replicas exceeding the policy's configured maximum; it does not prove four tasks can fit the cluster or that the new replicas reduce CPU. The scaling path and runtime observations still need their own checks.

For scale-down, take six replicas, observed 20, target 60 and minimum three. The raw result is ceil 2, then the minimum raises it to three. This protects the policy floor rather than a workload-specific availability SLO. If one of the three desired replicas cannot start, the equation does not independently compensate unless the provider's next sample and subsequent controller decisions reflect that situation.

There is also a scale-from-zero boundary. With current zero and MinReplicas zero, multiplication stays zero even if the supplied utilization is positive. This CPU formula has no request-queue activation signal. A service with a positive minimum can acquire a nonzero arithmetic base, but that is different from a complete scale-to-zero/activation mechanism. Do not infer features merely from a lower bound permitting zero elsewhere.

Bounds, Cooldown, and Rollout Exclusion

reconcileService checks active rollout before obtaining metrics. If GetServiceRollout succeeds with a nonterminal deployment, it records skip_rollout and leaves desired replicas alone. This preserves the deployment's ownership of its count transition. If the rollout lookup returns an error, hasActiveRollout returns false; the source does not treat every lookup failure as a fail-closed exclusion. That is a bounded behavior worth testing in a stronger composition.

Next, effectivePolicy substitutes a one-minute cooldown when the field is nonpositive. The controller compares its current clock with lastScaleAt for this service. A timestamp inside the interval produces skip_cooldown before calling the provider. At exactly the interval boundary, the less-than guard no longer skips. The clock is injected in tests, allowing a reproducible boundary without waiting in real time.

Unchanged source excerpt — internal/autoscaler/autoscaler.go:148–158:

```
        policy := effectivePolicy(service.Spec.Autoscaling)
        if last := c.lastScaleAt[service.ID]; !last.IsZero() &&
c.now().Sub(last) < policy.Cooldown {
                c.metrics.IncAutoscalerDecision("skip_cooldown")
                c.emit(ctx, service, events.TypeAutoscalerSkipped,
types.EventInfo, "autoscaling skipped during cooldown",
map[string]string{"reason": "cooldown"})
                return nil
        }
        utilization, err := c.provider.ServiceCPUUtilization(ctx,
service)
        if err != nil {
                c.metrics.IncAutoscalerDecision("skip_metrics")
                c.emit(ctx, service, events.TypeAutoscalerSkipped,
types.EventWarning, "autoscaling skipped because metrics are
unavailable", map[string]string{"reason": "metrics_unavailable"})
                return nil
```

lastScaleAt is a map on Controller. It is updated only after ScaleService succeeds, and a new controller starts with an empty map. A failed scale does not create a completed-action cooldown record. Reconstructing the controller loses the previous cooldown even if the supplied service object still exists. A fresh default server loses much more, because its MemoryService also starts anew; neither behavior supports a durable cooldown claim.

StabilizationWindow is declared and validated in AutoscalingPolicy, but this recommendation controller does not use it. There is no ring of previous recommendations or maximum-over-window downscale calculation here. Parsing and validating the duration establishes a field contract, not an enforced smoothing algorithm. An operator who tunes only that field should not expect this source edition's package to damp oscillation through it.

Policy concept	Implemented effect	Boundary
Minimum/ maximum	Clamp arithmetic recommendation	Does not reserve node resources

Policy concept	Implemented effect	Boundary
Cooldown	Skip decisions after successful scale within controller-local interval	Lost on controller reconstruction
Active rollout	Skip when lookup returns nonterminal deployment	Lookup errors treated as no active rollout
Stabilization window	Field exists and rejects negative durations	No recommendation-history enforcement

These distinctions are more useful than saying the controller “supports autoscaling policies” without naming which fields actually change behavior.

Fake Metrics and Package Tests

FakeMetricsProvider clones its initial map, protects updates with a mutex and supports Set and SetError. ServiceCPUUtilization returns an unavailable error for a service without a value. This makes it appropriate for deterministic decisions and missing-metrics tests; it is not a simulator of scrape lag, per-container utilization or production sampling noise.

TestAutoscalerScalesUpAndDown starts two replicas at target 50, supplies 100 and asserts four. It then supplies 25 after the configured cooldown and asserts two, along with one scale_up and one scale_down decision and two events. TestAutoscalerRespectsCooldown advances an injected clock by thirty seconds within a one-minute interval and keeps the first scaled count. Tests for clamping and active rollout complete the package's explicit contracts.

Provider errors produce skip_metrics and an autoscaler.skipped event, then return nil for that service. Thus a successful RunOnce can mean “metrics unavailable; no scale attempted.” Developers should not equate a nil return with convergence. Operators need decision labels/events and the provider's actual sample freshness to explain why desired replicas did not change.

RunOnce also skips disabled autoscaling and non-active services. A service error during reconciliation increments the controller error metric and returns from the pass, leaving later services for another pass. The Run loop logs errors and repeats at its cadence. Ordering and failure isolation therefore matter when extending the package to a large catalogue; the current implementation is not a proof that every service was evaluated during a successful or failed interval.

The source updates desired replicas through the same ScaleService seam used by API/CLI operations, preserving an important place for admission, maintenance and quota decisions. It does not bypass those rules by creating containers itself. A later integration must therefore account for a valid recommendation being rejected by the control plane, and must observe whether the approved desired count actually becomes available.

The Provider and Server Wiring Gaps

Searching the default composition root finds scheduler, rollout, node monitoring, GitOps and cron loops, but no autoscaler.NewController invocation. No production MetricsProvider is selected there. The presence of autoscaler metric declarations in the shared metrics package cannot activate a controller: declarations are storage for observations, not a scheduler of work.

A correct production provider would need to define utilization units, aggregation across task versions, stale or missing samples, authentication to its metrics source and bounded query latency. Consider a rollout containing old and new images: averaging whichever containers happen to appear can create a recommendation based on an incoherent population. A provider contract must state whether it measures current desired task identity and how it handles newly started replicas with no sample.

Then composition must select the provider, inject metrics/events, establish namespace coverage and supervise the controller. Leadership and durable cooldown are additional decisions, not automatic

consequences of constructing one goroutine. A separately tested teaching extension could demonstrate a Prometheus adapter or a local runner, but it must remain outside immutable evidence and cannot be described as the pinned server's behavior.

The Atlas failure timeline makes the gap concrete. At timeA, the YAML validates. At time B, the service is created with autoscaling enabled. At time C, CPU rises. The default binary has no autoscaler consumer, so no automatic replica change follows from that policy. This outcome is different from a running controller skipping stale metrics or cooldown. Diagnose the missing caller before searching for a bug in the recommendation equation.

A review should also keep demand and enforcement separate. Desired replica arithmetic can be sound while ordinary MemoryService service placement bypasses the planner's resource fits, and the active agent omits CPU/memory limit propagation. Autoscaling therefore cannot be used as evidence that node capacity or Docker cgroups are protected. Revisit those earlier field paths when evaluating a stronger composition.

Atlas Lab: Change Samples, Not Host Load

Use the disposable pinned source copy and Go 1.25. Existing package tests need neither Docker nor PostgreSQL. Their injected clock and FakeMetricsProvider exercise decisions without burning CPU on the actual machine.

```
go test ./internal/autoscaler -run 'TestAutoscaler(ScalesUpAndDown|
RespectsCooldown|ClampsMaxAndMinReplicas|SkipsActiveRollout)$' -
count=1 -v
```

The fresh package-test record includes the autoscaler decisions with caching disabled. The injected provider and clock establish deterministic package behavior; the default server still does not start this controller. For a separate teaching fixture, initialize Atlas with three replicas, min two, max four and target 60. Supply 90, check desired four, then supply 15 during cooldown and verify it remains four. Advance the injected clock

beyond cooldown and check the bounded reduction. This is a package composition, explicitly different from deploying policy through default server startup.

Add a missing-sample case and observe skip_metrics rather than inferring a failed process. Reconstruct the controller over the same service and show that lastScaleAt is empty; if you propose durable cooldown, test the different implementation independently. Compare a positive StabilizationWindow with zero using otherwise identical inputs and explain why this baseline makes no history-based difference.

Abort if the exercise starts a real stress workload, targets a non-lab service or silently patches startup to hide the missing integration. Cleanup discards only fake objects and owned temporary files. A later real provider experiment needs sample timestamps and explicit source credentials kept outside logs, plus a bounded workload and cluster scope. Passing the deterministic equation tests cannot establish production load handling or automatic scaling in a binary that never starts the controller.

Exercises

1. Compute Atlas's recommendation for three replicas, observed 90, target 60, minimum two and maximum four. Identify which step produces each intermediate value.
2. Explain why reconstructing a controller during cooldown can permit another decision even when the service count persists in a fixture.
3. What do a nonzero StabilizationWindow and a successful RunOnce prove about actual stabilization and metrics availability?
4. Draft an acceptance boundary for a production provider and explain why adding autoscaling YAML to the default server cannot satisfy it.

Worked answers preserve the distinction between calculation, policy enforcement, observation and executable composition.

Development history and source references

Original development prompts 54, 55 supply the requested scope. The inspected implementation history includes cf4766ccc7ae, b575252d6c5f. Their exported changed paths were compared with this chapter's implementation. These are topical associations; early prompts were retrospective and later work can share a commit. A tracker entry is not evidence that the described experiment ran.

- internal/autoscaler/autoscaler.go
- internal/autoscaler/autoscaler_test.go
- docs/AUTOSCALING.md
- docs/AUTOSCALING_DESIGN.md
- cmd/orch-server/main.go

Chapter 23 – Authentication, Authorization, and Node Trust

A viewer is allowed to inspect Atlas's deployment but must not replace its image. An agent is allowed to report the state of its assigned task but must not impersonate another node. Both requests reach the same server, yet they require different identities and different checks. Treating every bearer value as one universal token would erase the distinction between human authority, enrollment authority and ongoing worker ownership.

You will trace a user mutation and a node report through their separate authentication paths, inspect role selection and optional enablement, and explain the trust granted to Docker access. API contracts and agent report ownership are prerequisites. The source edition uses local HS256 user JWTs and short-lived node credentials; this chapter does not imply OAuth, an external identity provider or mutual TLS.

User JWTs and Role Checks

Default main parses static users and passes cfg.JWTSecret and the resulting user map into the API handler. userAuthMiddleware selects a namespace, validates it and asks requiredRole about the request. For a nonempty JWT secret on a protected route, it parses a bearer token, verifies its signature/claims and establishes the principal before the handler runs. A bad identity returns unauthorized; an authenticated but insufficient role returns forbidden.

Unchanged source excerpt — internal/api/api.go:1267–1275:

```
                required, ok := requiredRole(r)
                if !ok || s.jwtSecret == "" {
                        next.ServeHTTP(w, r)
                        return
                }
                token, err :=
auth.ParseBearer(r.Header.Get("Authorization"))
                if err != nil {
                        s.writeAuthError(w, r,
http.StatusUnauthorized, "unauthorized", "missing or invalid bearer
token")
                        return
```

The empty-secret condition is essential. Built-in server configuration leaves user JWT auth disabled unless configured, even though the development bootstrap token is nonempty. That distinction is a development default, not an assertion that every endpoint is publicly accessible: agent routes perform their own credential checks. A deployment review should record effective settings rather than infer user protection from the existence of auth code.

ValidateJWT accepts HS256 only, checks the HMAC, parses claims and requires a subject plus either a valid global role or namespace-role map. Each namespace role must be valid. Expiration is enforced when ExpiresAt is positive; a missing exp is not made mandatory by this implementation. IssuedAt is represented but not a complete replay-prevention protocol. The claims contract is narrower than a general-purpose identity-provider integration.

RoleForNamespace prefers a valid global role; otherwise it looks up the normalized namespace. Static users can override the token's role with a configured global role and clear namespace roles. This can be convenient for local administration, but it changes the authority model. An operator investigating an unexpected permission must inspect both token claims and static-user configuration, without printing the token itself.

Roles have ordered authority: viewer, operator and admin. Reading services/tasks/events/logs is viewer-level; deploying, scaling, rolling back and deleting services is operator-level; node drain/uncordon, namespace management, quota changes and audit access require admin in their named routes. Unrecognized protected paths fall back to admin rather than silently bypassing authorization. Exact route/method combinations, not friendly UI labels, define the check.

Agent Bootstrap Is a Different Trust Path

Registration accepts a configured bootstrap token through its header or bearer form. If the configured token is empty, this specific registration check permits enrollment; built-in configurations instead provide a known development value that must not be confused with a private production secret. The server issues a new random node credential after successful registration and stores only its hash and expiry in the node record.

The issuer reads thirty-two random bytes, base64url encodes the bearer value, calculates expiry and returns a separate record containing Hash and ExpiresAt. The default lifetime is fifteen minutes. This is a node credential represented as an opaque token, not a user JWT with viewer/operator/admin claims. The AgentIdentity interface is an extension seam; the current token format does not itself carry a certificate-bound node identity.

Unchanged source excerpt — internal/auth/agent.go:56–66:

```
func (i *TokenAgentCredentialIssuer) Validate(_ context.Context,
token string, record AgentCredentialRecord, now time.Time) error {
        if record.Revoked {
                return ErrInvalidToken
        }
        if record.Hash == "" || token == "" ||
HashAgentToken(token) != record.Hash {
                return ErrInvalidToken
        }
        if !record.ExpiresAt.IsZero() && !
now.UTC().Before(record.ExpiresAt.UTC()) {
                return ErrInvalidToken
        }
        return nil
```

Validation rejects revoked credentials, missing/mismatching hashes and expired records. The server fetches the specific NodeID's stored record before validating the presented token, binding the request to current node authority. Heartbeats issue another credential and replace the stored hash, so rotation changes which value is current. Raw credentials are returned only as needed to the registering/heartbeating agent and must not appear in publication evidence.

Rotation has an acknowledgement window. If the server records a new hash but the response carrying the new value is lost, the client may retain an older token that no longer matches. Source-based recovery expectations must account for that path rather than promise seamless indefinite retries. Expiry and revocation also need a re-enrollment policy; having an interface for future identity mechanisms does not implement one automatically.

RevokeNode sets the node's AgentRevoked flag. SetAgentCredential clears that flag when establishing a fresh credential. Thus revocation prevents use of the current record, but continued access to enrollment authority can matter for re-registration. It is not a permanently revoked certificate authority. Protect the bootstrap path separately from ordinary reporting.

Which Routes Require Which Identity

requiredRole excludes /v1/agent routes from user JWT handling. That means they proceed to agent-specific handlers, not that they are unprotected. Agent task retrieval and heartbeat call authorizeAgentCredential with the relevant node ID. Task status first validates that node credential, then ReportTaskStatus checks the task is assigned to the same node and the lifecycle permits the requested state. Authentication and assignment authorization remain distinct decisions.

A malicious or mistaken node token for node A cannot legitimately report a task owned by node B merely because both nodes are registered. The server also rejects status spellings outside the allowed agent states. A late running report against desired stopped must satisfy lifecycle guards rather than overwrite current authority. These checks protect record mutation; they do not physically fence an already-running isolated container.

User reads have their own subtleties. Secret GET access returns metadata rather than plaintext, and audit reads require admin. The Traefik provider endpoint is viewer-protected when user JWT is active, so an external consumer's polling must be configured accordingly. /healthz, /readyz and /metrics bypass user roles; their responses should not be described as privileged proof of full system health.

Server-to-agent log streaming uses the bootstrap-token authority on the agent listener, separate from the short-lived credential the agent presents to server routes. This asymmetry deserves explicit operational documentation. A user authorized to request logs relies on the server's proxy and node address handling; the worker's local listener has its own trust boundary. Do not copy the user's JWT into every downstream hop or assume node credentials protect an unrelated listener.

Request	Identity/authority	Additional boundary
User service rollout	Configured JWT and operator-or-higher role	Namespace, admission and operation ordering

Request	Identity/authority	Additional boundary
Node registration	Configured bootstrap token	Current enrollment policy
Task status report	Current credential for claimed NodeID	Task ownership and lifecycle guard
Server-to-agent logs	Bootstrap token at agent listener	Container/task selection and streaming scope

The table identifies current behavior, not a recommendation to share all credentials. It allows an operator to rotate the right authority and a developer to test each rejection at the correct layer.

Docker Access and Missing Mutual TLS

The trusted agent speaks to the Docker daemon. A caller with that socket authority can create powerful workloads, inspect environment and affect host resources. API admission can restrict submitted security options, but it does not turn the agent's daemon credential into a harmless read-only capability. Compromise of that trust path has a different consequence from compromise of a viewer JWT.

Native server/agent communication in this edition uses HTTP. Token possession authenticates requests according to server records; it does not encrypt transport or bind a peer certificate to a node. External TLS termination can protect selected links when configured, but mutual TLS, certificate issuance, renewal and revocation are not implemented merely because a future credential interface exists. State those missing responsibilities precisely instead of describing token auth as equivalent security.

Namespaces and roles are API boundaries. They do not create isolated Docker daemons, kernels or tenant networks. A legitimate operator can still deploy a workload within allowed policy, and an agent administrator can inspect local execution. Chapter 27 examines that isolation scope; Chapter 24 follows plaintext secrets to the same execution boundary.

The Atlas failure scenario combines these facts. User auth is enabled and viewer mutation is correctly forbidden. A worker's heartbeat then rotates its credential, but the response is lost. The next heartbeat uses the old credential and fails. That does not show the viewer bypassed RBAC; it shows a different identity's acknowledgement/recovery window. Incident analysis should retain rejection code, node/task ID and request time while redacting every bearer value.

Atlas Lab: Prove Rejection Without Publishing Tokens

Use Go 1.25 in the disposable pinned source copy. Existing tests generate their own synthetic credentials and verify outcomes without a real worker or external identity service. Avoid shell arguments containing actual owner tokens and do not copy signed fixture tokens into chapter listings.

```
go test ./internal/auth -run TestAgentCredential -count=1 -v
go test ./internal/api -run 'Test(UserAuth|
AgentRegisterRequiresBootstrapToken|AuditRequiresAdminRole)' -
count=1 -v
```

Expected observations include unauthorized missing/invalid user credentials, forbidden viewer mutation, accepted authorized reads and invalid/expired/revoked node credentials. The fresh package-test record records authentication and API assertions with caching disabled. The security laboratory separately exercises controlled wrong-node and authorization boundaries; neither result establishes mTLS or hostile-host isolation. For a separate route-matrix fixture, generate local viewer/operator/admin JWTs in memory, send requests through an httptest handler and record only method, path, status and whether state changed.

Include the empty-JWT-secret case explicitly so the test distinguishes disabled user auth from a valid identity. Test a node A credential on a node B task report and assert both rejection and unchanged task state. Add the rotation-response-loss case only as a labeled teaching

extension, preserving the original implementation. Any stronger recovery proposal requires its own tested protocol rather than changing expected output to fit the story.

Abort if a fixture points to a live server, captures an Authorization header or uses a real enrollment token. Cleanup discards generated identities and temporary logs after retaining sanitized outcome evidence. A pass establishes these handler/issuer contracts, not end-to-end TLS, Docker host containment or complete tenant isolation.

Exercises

1. Why can an empty user JWT secret disable role checks while agent task routes still reject a missing node credential?
2. A token grants a namespace viewer role, but static users configure its subject as operator. Which authority does this middleware select?
3. Trace the lost heartbeat-rotation response window. Which stored field changes and why can a previously valid credential stop working?
4. Explain why a protected viewer API and short-lived node tokens do not establish mutual TLS or containment after Docker socket compromise.

Worked answers identify current authorities and their limits rather than treating every successful authentication as universal trust.

Development history and source references

Original development prompts 18, 19, 51 supply the requested scope. The inspected implementation history includes 403a9fc9d900, 4f25e78c229b, a5cb0c5fb4ca. Their exported changed paths were compared with this chapter's implementation. These are topical associations; early prompts were retrospective and later work can share a commit. A tracker entry is not evidence that the described experiment ran.

- internal/auth/auth.go

- internal/auth/agent.go
- internal/auth/agent_test.go
- internal/api/api.go
- docs/SECURITY.md
- docs/SECURITY_REVIEW.md

Chapter 24 – Encrypted Secrets and Configuration Redaction

Atlas needs a database connection string, but its deployment manifest should name the secret rather than contain the real value. The control plane must protect its stored representation, supply plaintext to the authorized execution path and keep diagnostics from accidentally publishing it. These are three different obligations. Encrypting one map entry does not make the agent's environment encrypted, and printing a redacted configuration does not prove every log channel is safe.

You will trace namespace-bound secret resolution, inspect the actual local encryption envelope, compare configuration precedence with its redacted view and identify missing key-rotation mechanisms. Chapter 23 supplies identity scope; this chapter follows the value across the boundaries that remain after authentication. All examples use synthetic placeholders. No production secret belongs in a command argument, source file, exported transcript or chapter screenshot.

Secret References and Namespace Resolution

The upstream secrets-api example shows an environment key with secretRef pointing to a named secret and a separate ordinary environment value. The YAML parser turns that reference into SecretRefs rather than embedding its plaintext into ServiceSpec.Env. The service retains the desired reference; the current value is resolved later when assigned-task data is assembled.

CreateSecret checks name and envelope availability, encrypts the supplied plaintext, then takes the memory-service lock. It verifies the namespace exists and stores encrypted bytes, KeyID and timestamps under namespacedKey(namespace,name). Creating an existing name updates its encrypted representation and timestamp; creating a new one also checks quota. The returned SecretMetadata excludes the encrypted payload and plaintext.

GetSecret and ListSecrets read only metadata within the namespace context. A user allowed to inspect secret names does not thereby receive their values through these endpoints. TestSecretsCreateRetrieveMetadataAndRedactPlaintext explicitly searches create/get/list/event response bodies for its synthetic value and asserts absence. That test establishes specific response channels, not a universal property of every diagnostic and integration.

resolveEnvLocked starts by cloning literal Env, then resolves each reference using the service's namespace and the referenced name. Missing references or unavailable envelope produce errors without embedding the plaintext. Decryption errors report the reference name rather than the decrypted value. A resolved secret then overwrites the selected environment key in the cloned map.

Unchanged source excerpt — internal/controlplane/controlplane.go:821–837:

```
func (s *MemoryService) resolveEnvLocked(ctx context.Context,
namespaceName string, spec types.ServiceSpec) (map[string]string,
error) {
        env := cloneStringMap(spec.Env)
        for _, ref := range spec.SecretRefs {
                secret, ok := s.secrets[namespacedKey(namespaceName,
ref.Name)]
                if !ok {
                        return nil, fmt.Errorf("%w: referenced
secret %q not found", store.ErrInvalidState, ref.Name)
                }
                if s.envelope == nil {
                        return nil, fmt.Errorf("%w: secret envelope
is not configured", store.ErrInvalidState)
                }
                plaintext, err := s.envelope.Decrypt(ctx,
secrets.Ciphertext{KeyID: secret.KeyID, Data:
secret.EncryptedValue})
                if err != nil {
                        return nil, fmt.Errorf("%w: decrypt secret
%q", store.ErrInvalidState, ref.Name)
                }
                env[ref.EnvName()] = string(plaintext)
        }
        if len(env) == 0 {
```

The agent task DTO receives this resulting plaintext Env, and ensureTask supplies it to ContainerSpec. Docker then constructs the container environment. This is necessary execution delivery, not a failure of encrypted control-plane storage. It does mean the authorized agent, Docker daemon administrator and application process become plaintext trust boundaries. A GET secret-metadata result cannot be used to infer that those administrators are unable to read the value.

Updating a secret's map entry does not mutate the already-created container's environment in place. Existing runtime objects retain their creation-time configuration until a separate workload lifecycle changes them. A rotation plan must identify which tasks should restart and how

the application handles old/new credentials during transition. A successful secret update alone is not proof every workload now uses the replacement.

The Local Encryption Envelope

Envelope is a small interface with Encrypt and Decrypt methods. LocalEnvelope implements it using an AES key and a key identifier. NewLocalEnvelope rejects empty key material, decodes supported base64 AES key lengths or derives a 32 byte key by SHA-256 of the provided string. That derivation is deterministic key construction, not an implemented external key-management service or a password-stretching subsystem.

The key identifier contains a prefix and the first eight bytes of a SHA-256 digest of the key. It identifies which local key representation is expected; it is not a remote lookup handle. The implementation creates AES-GCM, obtains a fresh random nonce for each encryption, prepends that nonce to ciphertext and authenticates KeyID as associated data.

Unchanged source excerpt — internal/secrets/envelope.go:55–61:

```
        nonce := make([]byte, gcm.NonceSize())
        if _, err := io.ReadFull(rand.Reader, nonce); err != nil {
                return Ciphertext{}, fmt.Errorf("generate secret
nonce: %w", err)
        }
        data := append([]byte(nil), nonce...)
        data = gcm.Seal(data, nonce, plaintext, []byte(e.keyID))
        return Ciphertext{KeyID: e.keyID, Data: data}, nil
```

Decrypt rejects a nonempty mismatching KeyID, checks ciphertext length against nonce size and calls GCM.Open with the same associated data. Corrupt bytes or the wrong key fail rather than return plausible plaintext. TestLocalEnvelopeWrongKeyFails exercises that boundary; TestLocalEnvelopeEncryptDecrypt checks round-trip behavior and that stored bytes are not the literal synthetic value.

Those tests are deliberately narrow. They are not a cryptographic certification, a proof of secure key custody or a check that the current executable persists ciphertext across process restarts. In default MemoryService, encrypted secret records still belong to volatile maps. A new server process loses those records even if its local key remains available. Encryption and durability address different failures.

The envelope's key string can come from built-in development configuration unless the owner supplies another value. The default exists for local demonstration, not as a private deployment secret. A source review should name that default condition without publishing a user's effective key. Redacted configuration is appropriate evidence of which sensitive field exists, not of the secret's actual strength.

Configuration Precedence and Redaction

LoadServerWithFile starts with defaults, applies a file, applies environment, applies explicit overrides and validates the result. Agent and CLI loading follow the same layered idea through their own fields. Therefore explicit command settings outrank environment, which outranks file, which outranks defaults. The relevant question in an incident is the resulting effective setting, not which source an operator remembers editing.

Unchanged source excerpt — internal/config/config.go:95–107:

```
func LoadServerWithFile(path string, overrides ServerOverrides)
(ServerConfig, error) {
        cfg := defaultServer()
        if err := applyServerFile(&cfg, path); err != nil {
                return ServerConfig{}, err
        }
        if err := applyServerEnv(&cfg); err != nil {
                return ServerConfig{}, err
        }
        if err := applyServerOverrides(&cfg, overrides); err != nil
{
                return ServerConfig{}, err
        }
        return cfg, cfg.Validate()
}
```

String application helpers ignore some empty values, so “set an empty override to clear a secret” is not a universal supported operation. Registration-token environment selection also has explicit precedence over the older bootstrap setting. Inspect the exact loader path for the binary rather than assuming all environment names are interchangeable aliases.

ServerConfig.Redacted replaces the full database URL, bootstrap token, JWT secret, users and secret key with [REDACTED] when nonempty. It returns empty for empty values. It does not reveal a masked suffix or a credential hash that could become another identifier in evidence. Cluster policy has its own redacted view, while ordinary address/interval settings remain visible for diagnosis.

Agent redaction protects its bootstrap token, and CLI redaction protects its token. A value in a nonsensitive field is not magically sanitized by these views: server_url and node labels have their own meanings and should not be abused as secret storage. Audit metadata and notification text use different redaction routines, already examined in Chapter 15. There is no single blanket guarantee covering every arbitrary error string.

TestServerConfigPrecedence, TestAgentConfigPrecedence and TestCLIConfigPrecedence exercise actual loading order. TestRedactedConfigDoesNotExposeSecrets asserts the sensitive server keys equal the redaction sentinel. A meaningful additional diagnostic test should search for a unique synthetic canary across selected output channels and state exactly which channels were checked. An empty grep result from one config command is not proof application stdout never prints an injected environment variable.

The distinction mirrors runtime resources: declared fields and diagnostic output do not establish downstream enforcement. A separate captured Docker comparison in this edition found ordinary-agent Memory/NanoCPUs zero while a direct adapter populated the requested limits; that actual record is scoped to resource propagation, not secret confidentiality. Inspecting real runtime properties needs deliberate sanitized evidence, because a full Docker environment dump can itself disclose values.

Key Custody and Rotation Boundaries

The local key is an authority required to decrypt stored secret bytes. Losing it while retaining ciphertext is a different incident from losing the default process's entire memory state. Copying it into every transcript to make a lab reproducible would destroy the confidentiality boundary. Record a synthetic fixture identifier or a redacted presence check, and store any real operational material through an owner's separate secret-handling process.

There is no external KMS provider, remote envelope-data-key wrapping or automated key-ring rotation in this source. Changing the configured local key makes existing ciphertext's key identity incompatible. A complete rotation extension would need to retain old decryption authority, re-encrypt records under the new key, verify completion and define rollback/retirement. That is a protocol with failure windows, not a configuration rename.

For Atlas, imagine the server encrypts secret V1, the worker starts container A, then the operator updates the stored value to secret V2. The current container can still use V1. Now imagine changing the envelope key without migrating stored records: a later assigned-task resolution can fail even though secret metadata remains visible. These timelines demonstrate why stored representation, current reference resolution and runtime environment must be observed separately.

Deletion has a related boundary. Removing a secret record does not erase plaintext already delivered to an existing container. It may instead make later reference resolution fail. Do not describe API delete as revocation of every external copy. Application credential revocation belongs to the downstream service's authority, with workload restart and orchestration cleanup as additional steps.

A threat model should therefore enumerate metadata readers, authorized task recipients, Docker administrators, application logs, audit events and backup material separately. Namespace-scoped secret lookup limits which record is chosen; it does not create kernel isolation or prevent an administrator of the shared daemon from inspecting execution. The engineering decision is to protect each owner explicitly and avoid transferring one boundary's claim to another.

Atlas Lab: A Canary Without a Real Credential

Use the disposable pinned source copy with Go 1.25. The deterministic stage needs no database or Docker and covers encryption, wrong-key behavior, metadata secrecy, agent delivery and configuration precedence.

```
go test ./internal/secrets -run TestLocalEnvelope -count=1 -v
go test ./internal/config -run 'Test(ServerConfigPrecedence|
AgentConfigPrecedence|CLIConfigPrecedence|
RedactedConfigDoesNotExposeSecrets)$' -count=1 -v
go test ./internal/api -run
'Test(SecretsCreateRetrieveMetadataAndRedactPlaintext|
SecretInjectedIntoAgentTaskEnv)$' -count=1 -v
```

The fresh package-test record records envelope, configuration and API assertions with caching disabled. The security laboratory separately records a synthetic canary across permitted plaintext and forbidden metadata/log channels; it does not certify every possible application log. The injected sample value must be unique and obviously nonoperational. Generate it inside the fixture rather than place a real connection string in shell history. Retain only pass/fail channel names and sanitized diagnostics, not a full agent assignment payload containing plaintext.

For an optional owned Atlas deployment, add a reference using a supported teaching manifest and a disposable secret value. Prove metadata absence and intended application delivery through a boolean canary comparison, without printing the value or Docker's full environment. Compare a later secret update with the existing container identity to demonstrate that record updates and workload refresh are distinct. Use only run-owned namespace/service/container IDs.

Abort if the exercise reads real configuration secrets, logs bearer material or targets a user database. Cleanup removes the run-owned service before its secret reference and closes owned processes; preserve unrelated records. A useful result states which channel omitted the canary, which authorized recipient received it and which state remained volatile. It does not claim KMS integration, complete log sanitization or automatic credential revocation.

Exercises

1. Why can secret GET safely omit plaintext while an assigned task legitimately receives it? Identify both trust boundaries.
2. Explain the nonce, KeyID and associated-data roles in the local envelope. What does the wrong-key test establish and leave unproved?
3. A file sets key A, environment sets key B and explicit overrides set key C. Which key is selected, and why is the redacted print unsuitable for proving its exact value?

4. Updating a secret and changing the envelope key are two different operations. Design an observation plan that avoids assuming existing containers refresh or old ciphertext remains decryptable.

Worked answers connect confidentiality to concrete owners, configuration and recovery rather than a generic “encrypted at rest” slogan.

Development history and source references

Original development prompts 37, 44 supply the requested scope. The inspected implementation history includes ad738d1867ee, a4844c571a0d. Their exported changed paths were compared with this chapter's implementation. These are topical associations; early prompts were retrospective and later work can share a commit. A tracker entry is not evidence that the described experiment ran.

- internal/secrets/envelope.go
- internal/secrets/envelope_test.go
- internal/config/config.go
- internal/config/config_test.go
- docs/SECRETS.md
- docs/CONFIGURATION.md
- deployments/examples/secrets-api.yaml

Chapter 25 – Registry Credentials and Immutable Image Identity

Atlas API now needs an image from a private repository. Two questions must remain separate: which credential permits an agent to pull it, and which artifact should that pull retrieve? A password answers the first. A digest can identify the second. Neither fact alone establishes who built the artifact or whether its contents are safe.

This chapter builds on namespace-scoped secret handling and service versions. You will follow encrypted registry credentials into one assigned service task, calculate how the image parser separates a registry port from a tag, and distinguish supplied digest metadata from a registry lookup. The concrete failure window is a mutable tag changing between a reviewed manifest and an agent's pull. We will examine the implemented controls without claiming the default resolver contacts a registry.

Registry Authentication Without Plaintext History

A service manifest references a credential ID through imagePullSecret. It does not need to contain the registry password. CreateRegistryCredential validates the ID, registry and username, requires a configured envelope, then encrypts the password before adding the credential to MemoryService. The stored record contains ciphertext and a key ID; the public metadata representation excludes the encrypted password. Updating an existing ID replaces the encrypted material within its namespace. Claim ch25.credentials.

The namespace is part of the lookup key. A credential named atlas-pull in one namespace does not satisfy the same reference in another. Service creation rejects a missing namespace-local credential before recording

the service. This is a useful logical boundary, although it does not imply host-level tenant isolation. A node that receives authorized task execution material still becomes part of the trust boundary.

On assignment polling, ordinary service tasks take a specific path. The control plane finds the task's service, resolves its environment and resolves that service's imagePullSecret. The envelope decrypts the password into the transient RegistryAuth payload. The agent converts that payload into the runtime's authentication type, and PullImage passes the encoded credentials to the Docker client. Public service/event history should contain references and image metadata, not that transient password.

internal/controlplane/controlplane.go, lines 854–873 at the locked revision.

```
func (s *MemoryService) resolveRegistryAuthLocked(ctx
context.Context, namespaceName string, id string) (*RegistryAuth,
error) {
        id = strings.TrimSpace(id)
        if id == "" {
                return nil, nil
        }
        credential, ok := s.registries[namespacedKey(namespaceName,
id)]
        if !ok {
                return nil, fmt.Errorf("%w: registry credential %q
not found", store.ErrInvalidState, id)
        }
        if s.envelope == nil {
                return nil, fmt.Errorf("%w: secret envelope is not
configured", store.ErrInvalidState)
        }
        plaintext, err := s.envelope.Decrypt(ctx,
secrets.Ciphertext{KeyID: credential.KeyID, Data:
credential.EncryptedPassword})
        if err != nil {
                return nil, fmt.Errorf("%w: decrypt registry
credential %q", store.ErrInvalidState, id)
        }
        return &RegistryAuth{
                Username:      credential.Username,
                Password:      string(plaintext),
                ServerAddress: credential.Registry,
```

Encryption at rest does not remove plaintext from execution. The server must decrypt for this protocol, the authorized agent receives the credential, and the Engine performs the pull. A debug dump of assigned-task payloads can therefore leak credentials even when list endpoints are metadata-only. Capture IDs, registry host and success/error shape for the lab; never capture the actual password or encoded authorization header.

The envelope key must remain available to decrypt stored material. Losing it is different from losing a registry token, and rotating one requires a distinct plan from rotating the other. In the default application, memory-owned credential records also disappear with a new server

process. The existence of encrypted PostgreSQL fields is not automatic recovery of that state. Keep key custody, record durability and least-privilege registry access as separate operational decisions.

The service-specific path should not be generalized to every workload. The finite-job branch in ListAssignedTasks returns the task directly before the ordinary service environment/auth resolution. JobSpec does not expose imagePullSecret. A private-image Job therefore needs its own supported field and execution trace before documentation can promise the service credential behavior for it. Similar domain names do not establish identical delivery paths.

Image References, Tags, and Digests

The parser first trims the requested reference and rejects an empty value. It separates an optional digest at the at-sign, then distinguishes a tag colon from a registry-port colon by comparing the last colon with the last slash. Consequently registry.example.test:5000/team/atlas:v2 has a registry port and a tag; it is not a repository whose entire suffix is a tag. Claim ch25.parser.

A first path component containing a dot or colon, or equal to localhost, is recognized as an explicit registry. Otherwise docker.io is used. An unqualified single-component Docker Hub name receives the library/ prefix. If neither tag nor digest exists, the parser supplies latest. These transformations explain the metadata fields without requiring any network communication.

internal/image/reference.go, lines 30–50 at the locked revision.

```
        nameAndTag := requested
        digest := ""
        if before, after, found := strings.Cut(requested, "@");
found {
                if strings.Contains(after, "@") || !
digestPattern.MatchString(after) {
                        return types.ImageMetadata{},
fmt.Errorf("image digest is invalid")
                }
                nameAndTag = before
                digest = strings.ToLower(after)
        }

        lastSlash := strings.LastIndexByte(nameAndTag, '/')
        lastColon := strings.LastIndexByte(nameAndTag, ':')
        tag := ""
        namePart := nameAndTag
        if lastColon > lastSlash {
                tag = nameAndTag[lastColon+1:]
                namePart = nameAndTag[:lastColon]
                if tag == "" {
                        return types.ImageMetadata{},
fmt.Errorf("image tag is empty")
                }
        }
```

The accepted digest pattern is broader than a cryptographic proof of image existence. It checks an algorithm-shaped prefix and a sufficiently long hexadecimal suffix, lowercases the digest, and rejects malformed repeated at-signs. It does not download a manifest and verify those bytes. Nor should this small parser be advertised as a complete implementation of every registry reference grammar or algorithm policy.

For a clearly non-executing explanation, imagine registry.example.test/team/atlas@sha256 followed by 64 repeated hexadecimal characters. That can demonstrate the syntactic shape and a policy branch. It is not an available image, a verified registry artifact or permission to run a pull. The parser tests intentionally use such synthetic values. Label them as test data instead of copying them into an executable deployment manifest.

Input property	Parsed consequence	Remaining question
Bare image name	Docker Hub/library and latest default	What bytes will the tag identify?
Explicit registry port and tag	Registry and tag separated by slash position	Is that registry reachable and authorized?
Supplied digest	Digest metadata can be populated locally	Does the artifact exist and match policy?
Successful authenticated pull	Registry access worked for that artifact	Is its publisher trusted?

Keep RequestedImage for the operator's original request and the parsed registry/name/tag/digest for subsequent decisions. A rollout incident often requires both: the requested tag explains intent, while the actual immutable reference identifies the selected content. Showing only v2 can obscure whether two deployments used the same bytes; showing only a digest can obscure the user's original request.

Immutable Policy Checks and Rollout Identity

Admission supports require_digest, allow_mutable_tags and the latest-tag denial names. A digest requirement rejects metadata without a digest. Disallowing mutable tags likewise rejects digest-free references, while latest denial addresses an implicit or explicit latest reference. These policies overlap but are not interchangeable. A version-looking tag such as release-2026 remains mutable unless the registry independently protects it or the deployment uses a digest.

The default ClusterPolicy permits mutable tags. Configuring a stricter policy is therefore part of the experiment, not a baseline behavior to assume. The service path calls its configured image resolver, stores metadata on the candidate and runs admission. RequireDigest also marks metadata Pinned. The task creation helpers then use PinnedReference for pinned images, producing registry/name@digest rather than silently returning to the mutable request. Claim ch25.pinning.

The focused image test uses an injected fixedImageResolver and a synthetic digest. It creates a service, performs a rollout, inspects stored image metadata and checks a current-version task's pinned reference. The test also explicitly completes deployment state through a test helper before requesting the subsequent task creation. Its passing result establishes those field transformations and paths under the fixture; it is not an actual image pull or a completed physical rollout.

The real focused run passed parser, pinned-reference and rollout-metadata checks. No private registry account was contacted. This is enough to support the chapter's deterministic observation gate: supplied metadata is retained and policy-directed task construction uses the expected identity. A live private-pull gate remains separate and needs an owner-provided accessible image and actual available digest.

Consider the failure window when Atlas is requested as team/atlas:v2. A review approves that tag at one time, then the registry changes it before the worker pulls. With the default parser, the server has not resolved the tag to content, so the worker can receive different bytes from those the reviewer imagined. Requiring a digest closes that particular mutable-name ambiguity only when a real digest is supplied or resolved by an actual trusted resolver. It does not establish that the content was built honestly.

The runtime must also receive the same pinned string for both pull and create. Pulling one reference and constructing a container from another would reopen the ambiguity. Trace the task image into the agent's PullImage and ContainerSpec.Image rather than stopping at the service metadata. Container inspection can then compare the intended reference with the selected local image identity during a separately scoped live experiment.

Supplied Digests Versus Registry Resolution

ParserResolver.Resolve simply calls Parse. It has a context parameter to satisfy the interface, but does not use it for an HTTP registry request. The interface supports alternative resolvers, and the test injects one. That extensibility is not proof the default server performs tag-to-digest network resolution. Claim ch25.resolver.

A future remote resolver would need authentication, manifest/platform selection, bounded network behavior and a consistent decision about which digest it returns. An image index and a platform-specific manifest are different objects. A design would also need to carry the selected identity through service versions, rollout creation and worker execution. These are integration requirements, not missing details that can be filled with invented terminal output.

Digest identity is narrower than signature authenticity. An attacker can supply a digest identifying malicious content just as precisely as a trusted publisher identifies a legitimate release. Signing and vulnerability evaluation need their own trust policy and evidence. Chapter 29 treats the repository's proposals as design material; neither a successful Parse nor RequireDigest activates those proposed verifiers.

The development history makes this separation visible. fe41c80f83b62a3c8ea8430046acf088ec4014a1 adds registry credential support. f2836eabfff8e442d38984433f3e1f5c642667d1 batches image pinning with quotas and GitOps. The changed implementations support that history; they do not establish a separate deployment or verifier implementation for every original prompt title.

For the safe Atlas lab, use Go 1.25 and the disposable locked source copy. Run the following command there, with the environment contract's toolchain and caches. It has no Docker/database prerequisite and does not require a registry password.

```
go test ./internal/image ./internal/controlplane \
  -run 'TestParseImageReferences|TestPinnedReference|
TestRolloutStoresDigest|TestRegistry' \
  -count=1 -v
```

Expected observations are accepted/rejected reference cases and the fixed-resolver pinned task assertion. The linked execution record preserves actual command, timestamps and output. Cleanup is process exit and removal of any run-owned teaching files. For an optional live experiment, obtain a genuinely accessible artifact, record its verified digest and registry/platform context, and use a scoped robot credential. Do not turn synthetic test digests into pull commands.

Before a release, retain requested reference, selected digest, service version and actual task identity together. If a pull fails, distinguish missing credentials, denied access, absent artifact and transport failure rather than assuming all are digest-policy violations. That separation lets an operator change the credential without silently changing artifact identity, or change the artifact through a deliberate new rollout rather than mutating an old observation.

Exercises

1. Parse registry.example.test:5000/team/atlas:v2 into registry, name and tag. Why is the first colon not the tag separator?
2. A supplied synthetic digest passes the parser. List three operational claims this result does not establish.
3. Explain why requiring a digest with ParserResolver can reject a tag even when the registry could resolve it.
4. Design a private-image service pull observation without recording credentials, and explain why it cannot automatically certify private-image Jobs.

Worked answers connect syntax, policy and execution evidence. The operational boundary is precise: encrypted credential delivery and supplied digest pinning are implemented paths; automatic remote resolution, artifact authenticity and vulnerability safety need additional independently verified components.

Chapter 26 – Security Contexts, Admission, and Policy

Atlas API should run with fewer host privileges than the process managing Docker. That requirement has two parts: refuse dangerous workload intent, and carry accepted security fields into actual container creation. A policy rejection is useful evidence, but it is not the same observation as inspecting an Engine HostConfig. We need both field tracing and a clear limit on what a shared worker can isolate.

This chapter assumes service validation, agent DTOs and the Docker adapter. You will identify supported security-context fields, follow them through the ordinary service task path, and distinguish built-in admission from proposed external policy engines. The Atlas failure scenario is an unsafe manifest submitted through a second entry point after an operator believes the CLI already validated everything.

Container Security Fields That Reach Docker

For ordinary service tasks, ListAssignedTasks copies service.Spec.SecurityContext into the agent payload. ensureTask calls dockerSecurityContext and places the resulting value in ContainerSpec.Security. Unlike the missing resource-enforcement edge discussed earlier, these security fields have an explicit mapping through the agent. The mapping includes user, read-only root filesystem, privileged mode, added/dropped capabilities, host network, host PID and host-path mounts. Claim ch26.fields.

internal/agent/agent.go, lines 323–333 at the locked revision.

```
        return orchdocker.SecurityContext{
                User:                   context.User,
                ReadOnlyRootFilesystem:
context.ReadOnlyRootFilesystem,
                Privileged:             context.Privileged,
                CapAdd:                 append([]string(nil),
context.CapAdd...),
                CapDrop:                append([]string(nil),
context.CapDrop...),
                HostNetwork:            context.HostNetwork,
                HostPID:                context.HostPID,
                HostPathMounts:         mounts,
        }
}
```

The Engine adapter sets Config.User and HostConfig.Privileged, ReadonlyRootfs, CapAdd and CapDrop. It adds bind mounts, selects host network mode when requested and selects host PID mode when requested. The cap-drop helper defaults an empty configured list to NET_RAW. This is a source-derived default worth naming precisely rather than inferring the daemon's unrelated defaults. File access inside the image can still fail when the chosen user lacks permissions, so accepted configuration is not proof the application will start successfully.

A read-only root filesystem applies to that filesystem boundary. Writable bind mounts remain writable if their own mode permits it. Dropping capabilities narrows particular process privileges; it does not remove the agent's authority over the Docker socket or establish a separate kernel for each tenant. A non-root user can also execute vulnerable application code. These fields have concrete value without being a complete hostile-code sandbox.

Host-path mounts deserve particular care. The built-in rule cleans a path and requires equality with an allowed prefix or a child separated by a slash. A raw substring match would incorrectly accept similarly named sibling directories. The code's lexical prefix check is still not a general

guarantee against every host filesystem alias or administrator-selected dangerous path. Keep an allowlist narrow and understand what data is visible through an accepted mount.

For the safe lab, the security field path is inspected and fake creation tests supply deterministic coverage. Do not run a privileged container merely to demonstrate rejection: a rejected intent can be tested entirely in-process. Actual Engine inspection of a harmless accepted user/read-only/cap-drop combination would be a separate local observation. No such host inspection is inferred from an admission unit test.

Admission Order and Policy Decisions

Admission's Request includes actor, operation, namespace, spec and ClusterPolicy. Evaluate accumulates Violation values containing field, rule and message. Controller.Admit returns an Error with operation, namespace and that collection, and optionally records a rejection through an AuditSink. This creates an error shape that callers can inspect without parsing one human sentence. Claim ch26.admission.

internal/admission/admission.go, lines 75–87 at the locked revision.

```
func (controller *Controller) Admit(ctx context.Context, request
Request) error {
        violations := Evaluate(request.Spec, request.Policy)
        if len(violations) == 0 {
                return nil
        }
        if controller != nil && controller.sink != nil {
                controller.sink.RecordAdmissionRejection(ctx,
Rejection{
                        Actor: request.Actor, Operation:
request.Operation, Namespace: request.Namespace,
                        Service: request.Spec.Name, Violations:
violations,
                })
        }
        return &Error{Operation: request.Operation, Namespace:
request.Namespace, Violations: violations}
}
```

The evaluator covers required CPU/memory requests and limits, privilege/host namespaces, capabilities, host paths, approved registries, digest/mutable/latest rules, healthcheck presence and replica/public-port ceilings. It is deterministic code with supplied inputs. It does not execute Rego, contact a remote policy endpoint or scan an image. Actor and operation are useful request context even where the present evaluator's rules depend mainly on spec and static policy.

Follow the actual create order rather than relying exclusively on the design document. CreateService resolves image metadata, validates the candidate, calls admission, applies resource defaults through NormalizeServiceSpec, validates again, then enters the state lock for namespace/reference/quota checks and mutation. Consequently a require-resource rule can reject a missing explicit request before normalization supplies a default. The design's phrase normalized candidate must be qualified by this real sequence.

A structurally invalid manifest can fail before admission and need not produce a policy-violation collection. A structurally valid but disallowed privileged request reaches the policy branch. A missing secret reference or quota exhaustion occurs later and has its own error type. Those are different acceptance boundaries. Keep tests for each rather than expecting every rejected manifest to produce the same rule identifier.

The API converts admission.Error into a structured rejection with violations, operation and namespace. Human-facing Error() emphasizes the first violation, while the structured collection preserves multiple findings. A UI should display stable field/rule values and useful messages, not treat message punctuation as a machine contract. Rejection auditing records rule context rather than raw secret values.

The failure window is a caller that bypasses the checked mutation method. Local CLI parsing alone cannot protect an HTTP client that submits a candidate directly. Conversely, a central method protects several ordinary entry points only if those callers actually invoke it. Every new mutation path needs the same trace; the presence of an admission package does not automatically intercept arbitrary map changes.

The Same Checks Across Entry Points

The CLI-assisted deployment parses supported YAML and submits the resulting service request. The API calls the control-plane service method. GitOps receives cli.ParseDeploy as its parser and applies through ApplyService, which uses the checked create/update paths. These shared calls are stronger evidence than three independent copies of loosely equivalent validation code. They also make one policy change visible to several ordinary entry points. Claim ch26.entry-points.

Scale and rollout construct candidates from current service intent, change the relevant field and invoke admission before committing the operation. The focused control-plane test creates an allowed service, rejects a scale above the policy maximum and rejects a latest-image rollout. It checks structured rule information for scale and failure audit records for both operations. The actual focused suite passed these rejection assertions alongside policy/admission tests.

That result should be stated at its actual scope. It demonstrates the selected built-in rules and tested mutation paths using in-process state. It does not prove every conceivable endpoint or future controller cannot bypass admission. Nor does it prove accepted resource limits reach Docker: chapter 5's field-path concern remains even if policy requires the declaration. Presence validation and runtime propagation answer different questions.

GitOps has an additional exception that matters for this boundary. Its ordinary fetch/parse/apply path uses shared admission, but MemoryService.GitOpsStatus can auto-revert a drifted service by restoring stored DesiredSpec directly. That branch does not call Admit again in the inspected method. Chapter 30 examines its implications. Therefore describe the checked ordinary apply path precisely and avoid a universal claim that every internal state restoration reruns current policy.

Finite jobs also have a distinct CreateJob path. Their JobSpec validation and resource defaulting are not the ordinary ServiceSpec admission sequence. A service policy test cannot certify every Job field or workload

type. If an organization needs the same policy across all workloads, first define the common candidate contract, identify the missing paths, and add independently tested checks rather than broadening the documentation by analogy.

Boundary	What it checks	What it does not establish
YAML/domain validation	Supported structure and field relationships	Organization policy compliance
Built-in admission	Candidate against static policy rules	Runtime field propagation
Agent/Engine mapping	Accepted security options reach create arguments	Full host isolation
Logical namespace access	Selected object visibility and roles	Independent worker kernel/network

The table is a map of responsibilities, not a ranking of security strength. An attacker can exploit a gap between any two rows. For Atlas, record both the rejected candidate's stable violation and the source path of the accepted security field. That makes the evidence actionable: one test protects admission, another protects propagation, and the operational deployment still owns the host trust decision.

What Admission Does Not Isolate

The worker agent can change Docker resources, and the daemon is a powerful host boundary. A policy that denies privileged mode prevents one dangerous declaration; it does not revoke the agent's daemon authority. Shared networking, kernel behavior, mounted data and resource competition remain relevant. Logical namespaces and declarative constraints should not be sold as certification for mutually hostile tenants.

The repository's policy-engine design proposes dynamic providers, revisioned bundles and an OPA-compatible evaluator boundary. Those ideas are useful architecture work. The inspected MVP contains built-in Go rules and no active external Rego execution in this path. A future

adapter would need bounded requests, authenticated policy material, explicit outage behavior and versioned decision evidence. None of those guarantees follows from writing an interface name in a design document.

A fail-closed decision can protect the acceptance boundary while reducing deployment availability when its dependency is unavailable. A fail-open decision can preserve deployments while weakening the intended control. Such tradeoffs must be explicit, scoped and tested. The present local evaluator avoids that remote dependency, but it still relies on a correctly configured static policy and on all relevant callers using it.

History distinguishes implementation from proposal. a140c4190af17969d207b5738cc7527e4ad96075 adds container security policy and related runtime fields. 9e78fed31df03d7a985a9f4ceae4770739b05014 batches admission/API policy and namespaces. Their changed hunks explain actual evolution; the broader policy design remains classified separately.

Reproduce the safe Atlas rejection lab from the disposable source module with Go 1.25 and the environment contract. No Docker, PostgreSQL or external policy server is needed. The checked command is:

```
go test ./internal/policy ./internal/admission ./internal/
controlplane \
  -run 'Test.*(Policy|Admission|Security)' -count=1 -v
```

Before execution, predict that a privileged candidate under default policy is denied, a scale beyond the configured maximum yields replicas.maximum, and a latest rollout under the relevant rule fails without changing accepted intent. Read the actual test assertions rather than assuming every possible rule was selected by the expression. The linked log records the cases that ran. Cleanup is process exit; no unsafe container was created to obtain a denial.

For an optional extension, submit the same valid-but-disallowed service through CLI, direct API and a local committed GitOps fixture. Assert matching stable rules and unchanged service intent, while preserving

distinct request/actor context. Use disposable identities and no secret-bearing bodies. Also include a previously allowed stored GitOps desired state under changed policy, because it exposes the internal restoration exception rather than merely repeating the happy path.

Exercises

1. Trace read-only root filesystem from ServiceSpec to the Engine create arguments. Why does a writable bind mount remain a separate concern?
2. Why can require_resource_requests reject a candidate before defaults exist? Identify the real ordering rather than quoting the design's normalized-candidate phrase.
3. Compare a malformed YAML request, a privileged policy denial and a quota rejection. Which should produce admission violations?
4. Explain why passing scale/rollout admission tests does not certify every Job or GitOps restoration path, or establish a hostile-tenant sandbox.

Worked answers connect the field path and acceptance sequence. The defensible promise is built-in rejection on inspected mutation paths plus explicit Engine mappings, with host trust and unintegrated policy proposals kept visible.

Chapter 27 – Namespaces and the Limits of Multi-Tenancy

Two teams want a service called atlas-api. Renaming every object globally would make ownership awkward, while allowing every request to see every ID would undermine the separation. Orch namespaces solve a specific part of that problem: selecting logical object scope and authorization context. They do not create separate Docker hosts or independent networking domains.

This chapter assumes authentication, secret references and the ordinary service path. You will trace the namespace from HTTP header or CLI selection into domain lookups, distinguish an invisible cross-namespace ID from a forbidden namespace request, and examine the data loss hidden in a namespace down migration. The Atlas lab deliberately proves logical isolation only, using in-process services with no real tenant workloads.

Namespace Context and Local Names

Normalize trims whitespace, lowercases the name and maps an empty value to default. Validate accepts one to 63 lowercase alphanumeric characters or hyphens with alphanumeric ends. The context helpers carry the normalized selection under a private key. FromContext returns a normalized value, including default when no selection exists. Claim ch27.context.

The API selects X-Orch-Namespace first when nonempty, otherwise the namespace query parameter, and then normalizes it. Middleware validates the selection and places it in request context before reaching protected operations. The CLI's namespace option is sent through its client configuration to the corresponding HTTP selection. A CLI flag is not an independent permission grant: the server still authenticates the caller and checks its role for the selected scope.

internal/api/api.go, lines 1303–1308 at the locked revision.

```
func selectedNamespace(r *http.Request) string {
        if value := strings.TrimSpace(r.Header.Get("X-Orch-
Namespace")); value != "" {
                return namespace.Normalize(value)
        }
        return namespace.Normalize(r.URL.Query().Get("namespace"))
}
```

Context selection has an important internal nuance. Selected distinguishes an explicitly supplied context value from an absent one. Matches allows all object namespaces when selection was not explicit, but compares normalized names when it was. This supports internal operations that need broader inventory. It also means direct library calls with context.Background are not equivalent to authenticated HTTP requests, whose middleware supplies explicit scope. Do not use an unscoped internal list as evidence that the API leaks every tenant.

The MemoryService service uniqueness check combines namespace and name. The namespace migration makes the corresponding SQL constraint UNIQUE(namespace, name), and changes secret/registry primary keys to include namespace. Thus team-a/atlas-api and team-b/atlas-api can be separate objects with different IDs. Names become convenient local handles; IDs remain distinct object identities and still require scoped access checks.

A practical incident bundle should record both namespace and ID. An alert saying only atlas-api can point to the wrong team's service; a copied ID without context can return not-found to a properly scoped operator. Adding namespace to URLs or CLI output improves operational clarity, but the authoritative separation still belongs in server lookup predicates rather than display conventions.

The focused source tests use the same name api in team-a and team-b, with zero replicas. They verify distinct IDs and filtered listing without starting Docker containers. The name is incidental to the invariant; the

Atlas exercise applies the same contract to atlas-api. A successful create here proves stored logical objects, not two physically running application instances.

Cross-Namespace IDs and Secret References

GetService checks both whether the ID exists and whether namespace.Matches permits that object. When an explicit team-a context asks for a team-b ID, it returns ErrNotFound. This prevents the object lookup from disclosing a distinct exists-but-in-another-namespace outcome. It is a domain visibility decision, not proof of secrecy against every timing or operational side channel. Claim ch27.visibility.

internal/namespace/context.go, lines 44–55 at the locked revision.

```
func Selected(ctx context.Context) (string, bool) {
        if ctx == nil {
                return Default, false
        }
        name, ok := ctx.Value(contextKey{}).(string)
        return Normalize(name), ok
}

func Matches(ctx context.Context, objectNamespace string) bool {
        selected, explicit := Selected(ctx)
        return !explicit || selected == Normalize(objectNamespace)
}
```

At the HTTP authorization layer, a caller lacking any qualifying role for the selected namespace can instead receive forbidden before object lookup. These cases must not be conflated. A viewer allowed in team-a but selecting team-b is denied at the role boundary; an authorized request within team-a for an invisible team-b ID reaches scoped lookup and looks absent. Tests should state which boundary they exercise.

Secret resolution uses namespacedKey with the workload's namespace and secret name. Service creation checks the reference there, and assignment resolution decrypts that same namespace-local record. A secret database in team-a therefore cannot satisfy team-b's reference

merely because the text name matches. The focused cross-namespace secret test expects an invalid-state rejection for the missing local reference.

Registry credentials follow the same namespace-local key pattern for ordinary service tasks. This limits accidental cross-team references, but an administrator with cluster-wide authority can still act across scopes. A credential intentionally recreated in two namespaces is two records even if it ultimately authenticates to the same registry account. Keep external account ownership separate from internal namespaced identity.

The executed focused suite passed namespace object and secret-reference isolation cases, plus API namespace-role tests. It uses in-process handlers and disposable test signing material. It does not contact a production identity provider, run network isolation probes or certify a tenant security boundary.

The Atlas failure window is a copied service ID submitted under the wrong namespace. An operator might interpret not-found as deletion, then create a new service with the same local name. Before concluding state was lost, verify the selected header/flag, caller role and object namespace. Conversely, do not add a globally unscoped fallback lookup merely to make that operator error more convenient; doing so changes the visibility contract.

Namespace Roles and Cluster Administration

Principal can contain a global Role or a NamespaceRoles map. RoleForNamespace returns a valid global role first; otherwise it looks up the normalized namespace in the map. The role hierarchy distinguishes viewer, operator and admin capabilities in the API. Namespace selection and role selection must therefore be evaluated together. Claim ch27.roles.

A global viewer has a broader scope than a viewer entry only for team-a, even if both can read a service in team-a. Similarly, a namespace-scoped admin entry should not silently become cluster administration. The middleware has special handling for namespace-management routes so

a role map alone does not grant the global namespace operation. The actual required role for each route still matters; a claim shape does not replace the route's authorization check.

Request	Relevant boundary	Expected interpretation
Team-a viewer selects team-a	Namespace role and read permission	Authorized scope
Same viewer selects team-b without role	Authorization before domain lookup	Forbidden
Team-a scoped lookup uses team-b ID	Object namespace match	Not found
Namespace-only role attempts cluster namespace management	Cluster role requirement	No automatic cluster authority

The table describes source/test paths, not a universal response for requests with invalid tokens or malformed names. Those fail earlier. Tests should retain valid authentication while varying only namespace scope when measuring this distinction, otherwise an unauthorized response can hide whether the scoped object check was reached.

Migration 000011 introduces a default namespace and assigns existing relevant rows to it. It adds namespace columns and indexes to services, tasks, deployments, events, secrets, registry credentials and audit logs. Foreign-key relationships and uniqueness become part of the stored logical model. Applying that migration is still a database operation separate from the default server's memory-owned state.

The down migration is destructive. It explicitly deletes non-default rows from several tables, including services, tasks, deployments, events, secrets, registry credentials and audit history, before removing namespace columns and restoring older uniqueness. It also deletes related service versions. This is not a reversible transformation preserving both teams' data under globally unique names. The appendix recovery runbook must require backup/restore review rather than presenting down/up as harmless housekeeping. Claim ch27.rollback.

A schema downgrade can also affect evidence: removing non-default audit rows destroys part of the incident record. Keep operational exports and database backups according to the actual retention/recovery plan before considering such a change. A passing migration build does not mean a production rollback has acceptable data semantics. The safe chapter lab does not execute this destructive downgrade.

Logical Isolation on Shared Workers

Namespaces organize API state, local names, roles, secrets and selected accounting. Nodes remain shared cluster resources. The agent can receive tasks across namespaces, and the Docker daemon still has host authority. A logical boundary that hides a service ID does not stop one host process from exploiting kernel vulnerabilities, exhausting an unbounded resource or reaching an exposed port.

The ordinary service path's scheduler/resource limitations are especially relevant. Namespace quotas account declared requests, while actual host enforcement needs a complete field path. A namespace name is not a cgroup, a network policy or a disk allocation. Do not infer physical isolation from a successful accounting rejection or an HTTP404.

Volume naming needs a separate review as well. Namespaced metadata can distinguish equal local names, but the Docker adapter ultimately receives a named volume string. If that string does not incorporate a unique namespace/volume identity, two logical names may share a physical daemon namespace. Chapter 32 examines the exact mapping. Cross-namespace secret tests do not cover this storage path.

This leads to a useful operational choice: treat shared workers as a common trust domain until stronger isolation mechanisms are independently implemented and tested. Separate hostile workloads at a boundary that matches the threat model. The book does not prescribe a particular product or claim certification; it identifies what the source actually enforces so the reader can avoid relying on absent mechanisms.

The real history is 9e78fed31df03d7a985a9f4ceae4770739b05014, which batches namespace/API policy work. Original prompt 65 records requested namespace intent. The current code and tests establish the logical checks; a tracking entry or design paragraph cannot expand them into physical multi-tenancy.

For the safe Atlas lab, use the disposable source module, pinned Go 1.25 and no server/database/Docker process. Predict two distinct same-name services, one visible service per explicit team context, a cross-scope ID looking absent and a missing local secret reference being rejected. Run:

```
go test ./internal/namespace ./internal/auth ./internal/
controlplane ./internal/api \
  -run 'TestNamespace|TestSecretReferencesCannotCrossNamespace' -
count=1 -v
```

The execution log records actual selected cases. Inspect their request headers and assertions; a test that only checks different IDs would not prove list filtering or role behavior. Cleanup is test process exit and temporary-file removal. No real tenant data or credentials are needed.

An extension can create two Atlas-named services through an in-process API handler with an explicitly scoped principal. Capture status codes and sanitized IDs, then vary one request dimension at a time: scope, object ID or role. Keep the expected forbidden/not-found distinction explicit. A live shared-host penetration or storage-isolation claim is outside this lab and requires a separate authorized threat model and evidence.

Exercises

1. Explain why context.Background can permit a broader internal Matches result than an explicit default namespace context.
2. Distinguish forbidden namespace selection from a not-found cross-namespace object ID. Which layer decides each?

3. Trace a same-named secret reference in team-b when only team-a owns that secret. What result should the service method produce?
4. An operator proposes the namespace down migration as a reversible cleanup. Identify data and evidence at risk, then explain why logical isolation still does not establish hostile-worker isolation.

Worked answers retain the layer distinctions and destructive rollback semantics. Namespace controls are valuable when their scope stays explicit: object visibility and authorization are implemented; independent host, network and storage isolation are separate guarantees.

Chapter 28 – Quotas and Usage Accounting

Atlas API requests CPU and memory for each replica. A namespace operator wants to prevent uncontrolled growth and later understand its declared footprint. Both tasks require arithmetic, but neither is the same as measuring CPU consumption or charging money. An accepted quota check can bound an organization's declared intent while still leaving runtime enforcement and commercial accounting unresolved.

This chapter assumes resource quantities, namespaces and guarded mutations. You will calculate requested usage, explain why concurrent changes need one acceptance boundary, and follow snapshots into CSV export. The Atlas failure scenario is two simultaneous scale requests that each appear safe against the same old total. A second failure is interpreting summed snapshots as an invoice-ready duration measure.

Requested Resource Arithmetic

ResourceUsage includes service count, replica count, CPU millicores, memory bytes, public-port declarations, secrets and registry credentials. addServiceUsage adds requests multiplied by replicas for CPU/memory. It increments service count once per service and public ports once per positive published-port declaration. It does not multiply public-port declarations by replica count. Claim ch28.arithmetic.

internal/controlplane/controlplane.go, lines 2177–2187 at the locked revision.

```
func addServiceUsage(usage *types.ResourceUsage, spec
types.ServiceSpec) {
        usage.Services++
        usage.Replicas += spec.Replicas
        usage.CPUMillicores +=
spec.ResourceRequirements.Requests.CPU * int64(spec.Replicas)
        usage.MemoryBytes +=
spec.ResourceRequirements.Requests.Memory * int64(spec.Replicas)
        for _, port := range spec.Ports {
                if port.PublishedPort > 0 {
                        usage.PublicPorts++
                }
        }
}
```

Suppose Atlas has two replicas, each requesting 250m CPU and 128Mi memory, and its spec declares one positive published TCP port. The service contributes one service, two replicas, 500 millicores, 268435456 memory bytes and one public-port declaration. A dynamic declaration with published_port zero does not add that same positive-port count at this boundary. This is declared service usage, not the number of actual host socket bindings.

Resource limits are not the quantities summed by this helper. A service requesting 250m with a larger limit contributes250m per replica to requested accounting. Likewise, an idle process and a busy process with the same declaration produce the same requested total. These distinctions are necessary when capacity planning, quota acceptance and observed utilization appear in one dashboard.

Deleted services are excluded from the usage inventory. A service scaled to zero can still contribute one service and its declared positive public ports, while contributing no requested CPU/memory through replicas. That behavior can surprise an operator who expects every dimension to become zero at scale zero. The code defines each dimension independently; document them before using an aggregate for decisions.

ResourceQuota uses zero as unlimited for each checked dimension. quota.Validate rejects negative configured limits. Check compares only positive limits against the projected usage and returns the first exceeded dimension in a fixed order. A zero CPU maximum is therefore not a request to prohibit CPU use; it disables that particular ceiling. Choose an explicit positive bound when the intended policy is finite.

StorageClaims appears in usage snapshots, but the current quota type/check does not add a corresponding storage-capacity limit. Counting claims is not measuring bytes stored, provisioned capacity or I/O. The source's implemented dimensions should not be broadened because a report contains a storage-related column.

Atomic Quotas Under Concurrent Changes

MemoryService performs usage recomputation, quota checking and relevant state mutation under its mutex. For scale, it builds a candidate and computes usage with the old service excluded and the replacement included. Counting both would wrongly double-charge the update; counting neither would permit under-accounting. The lock is what makes the check and mutation one local acceptance boundary. Claim ch28.concurrent-memory.

The focused concurrency fixture starts two one-replica services under a three-replica quota. Two goroutines attempt to scale different services to two at the same time. Each change would be valid alone: total three. Both succeeding would produce total four. The executed test asserts exactly one success and one typed quota denial. It proves the MemoryService boundary for that scenario, not distributed coordination across independent server processes.

PostgreSQL has a distinct implementation. lockQuota acquires a transaction advisory lock derived from the namespace, and lockAndCheckQuota loads current usage plus the replacement/additions before checking. Store mutations using that helper execute within the

transaction-scoped store. The lock lifetime and complete caller path matter: a helper called outside the encompassing transaction would not provide the intended serialized mutation. Claim ch28.concurrent-store.

internal/store/postgres.go, lines 149–165 at the locked revision.

```
func lockQuota(ctx context.Context, db postgresDB, namespaceName
string) error {
        _, err := db.Exec(ctx, `SELECT
pg_advisory_xact_lock(hashtextextended($1, 0))`, namespaceName)
        return mapPostgresError(err)
}

func lockAndCheckQuota(ctx context.Context, db postgresDB,
namespaceName string, replaceID types.ServiceID, replacement
*types.ServiceSpec, addSecrets int, addRegistries int) error {
        if err := lockQuota(ctx, db, namespaceName); err != nil {
                return err
        }
        value, usage, err := loadQuotaAndUsage(ctx, db,
namespaceName, replaceID, replacement, false)
        if err != nil {
                return err
        }
        usage.Secrets += addSecrets
        usage.RegistryCredentials += addRegistries
        return quota.Check(value, usage)
}
```

The default server remains MemoryService, so do not present this SQL protocol as the live executable's quota backend. The untouched PostgreSQL integration fixture also has the schema mismatch recorded earlier. Its failure is not a passing quota race. Source inspection supports the intended transaction/lock mechanism; a separate actual all-schema quota contention experiment would be needed before recording that specific database behavior as executed.

Typed quota.Error contains namespace, resource, limit, used and requested. In the current Check helper, Used and Requested are both set to the projected total being checked. They do not necessarily mean old

total and incremental delta. If Atlas scale would project four replicas under a limit of three, the consumer should interpret the actual fields rather than subtracting a fictional delta.

The API returns a structured conflict for quota exhaustion. This differs from admission policy rejection and from malformed domain input. A client can explain which resource exceeded which limit without parsing English Error() text. The focused executed suite passed MemoryService allow/deny, concurrent scale, secret/credential quota and structured API conflict assertions.

The failure window is check-then-write without shared serialization. Reading total two in two independent requests and both approving an increment to three would violate the intended combined maximum. The mutex fixture catches this locally. Across processes, the SQL path must hold its namespace lock through the actual write; across other mutation paths, each path must participate. Jobs and GitOps restoration deserve separate inventories rather than being assumed covered by a service-scale test.

Usage Snapshots and CSV Units

CaptureUsageSnapshots locks MemoryService, computes current requested service usage for each namespace, counts volume claims and accumulates task runtime from StartedAt to FinishedAt or the current time. It appends one snapshot per namespace and sorts the returned snapshots by namespace. The timestamp is the capture instant, not a billing-period boundary. Claim ch28.snapshots.

TaskRuntimeSeconds is particularly important. Each capture sums the lifetime-to-date duration of retained started tasks. It is not an incremental duration since the previous snapshot. A task started at12:00 and still running contributes60 seconds at12:01 and 120 seconds at12:02. Summing those two snapshots produces180 in report totals even though only120 seconds elapsed since start. This is how the current aggregation behaves; it must not be labeled integrated task-seconds for that interval.

GetUsageReport defaults its end to now and start to24 hours earlier when omitted. It selects snapshots whose timestamps fall within the supplied bounds, including equal endpoints, and sums each selected field into Totals. CPU and memory totals are therefore sums of sampled requested quantities, not time-weighted core-hours or byte-seconds. Sampling frequency changes those sums even if actual intent is constant.

The CLI export writes snapshot rows with a stable header: namespace, timestamp, cpu_millicores, memory_bytes, replicas, services, task_runtime_seconds, public_ports and storage_claims. Timestamp uses RFC3339 and runtime seconds are formatted to three decimal places. The export exposes observations for downstream analysis; it does not silently normalize them into commercial units.

Column	Unit/meaning	Unsafe interpretation
cpu_millicores	Requested CPU at capture	Measured CPU time
memory_bytes	Requested memory at capture	Peak resident memory
task_runtime_seconds	Retained task lifetime accumulated at capture	Incremental billable seconds
public_ports	Positive service declarations	Actual host bindings across replicas
storage_claims	Namespace claim count	Stored bytes or replicated capacity

The actual snapshot aggregation test creates a two-replica service with 250m requests and one claim. It checks500 millicores and one storage claim, then verifies report service/replica totals. That is executed evidence for those assertions. It does not validate every export consumer, arbitrary time intervals or a complete accounting model. Preserve the distinction between inspected CSV formatting and the specific test cases that ran.

A robust downstream calculation would first decide which quantity it needs. To estimate reserved CPU-time, integrate the requested CPU value over well-defined intervals with a missing-sample policy. To

estimate actual CPU-time, collect a different trustworthy measurement. To charge task runtime, avoid summing repeated cumulative lifetimes and define how deletion, retries and overlap affect identity. These are additional designs, not automatic meanings of the existing totals.

An Export Boundary, Not a Billing System

The repository supplies operational quota decisions and usage reporting. It does not define prices, money, taxation, invoices, payment collection, customer contracts or dispute resolution. The original tenant-billing-export prompt motivates the report; its title is not evidence of a commercial billing engine. Calling the output a usage export accurately describes its current role.

Memory snapshots disappear with the default server's process state. Retention can also remove source task history or old observations according to its own rules. An operator relying on external historical analysis must arrange collection and storage outside this memory lifetime, while avoiding secrets in exported records. A present API response does not promise reconstruction of every past sample after a restart.

The known report dimensions also leave gaps. Pending desired replicas contribute requested usage even if no container has started. Job attempts and local storage need their own accounting interpretation. A claim count says nothing about whether a volume remains after workload deletion. These are reasons to document definitions, not to guess additional columns or invent measured values.

History starts with f2836eabfff8e442d38984433f3e1f5c642667d1, which batches quotas, image pinning and GitOps. 7985b9ebe276f8447e1b8abfceb0d6b746ea6f2d includes later operational usage/report work. Their actual changed code provides the historical connection; it does not certify invoice accuracy or deployment adoption.

For the safe Atlas lab, use only the disposable module and Go 1.25. No Docker or database is needed for the selected MemoryService/API tests. Predict one winner and one denial for concurrent scale, a projected usage error at the bound, and 500 millicores for the two-replica snapshot. Run:

```
go test ./internal/quota ./internal/controlplane ./internal/api \
  -run 'TestResourceQuota|TestConcurrentScaleCannotExceedQuota|
TestSecretAndRegistryCredentialQuotas|TestUsageSnapshotAggregation|
TestQuotaAPI' \
  -count=1 -v
```

The linked execution record preserves actual output and selected cases. Cleanup is process exit. An optional export exercise can use a separately owned in-process server and a bounded date range, then compare its CSV header and units with the code. Do not point the chapter at a production account or treat a synthetic task duration as a real host measurement.

Exercises

1. Calculate CPU, memory, service, replica and public-port usage for the two-replica Atlas example. Which dimension is not multiplied by replicas?
2. Explain why two individually valid scale requests cannot both succeed under the three-replica fixture. Name the MemoryService and PostgreSQL mechanisms separately.
3. A task contributes60 seconds and then120 seconds in two snapshots. What does the current report total, and why is that not120 billable seconds?
4. Design a bounded operational export interpretation that avoids claiming measured utilization, storage bytes or invoices from the present columns.

Worked answers show the arithmetic and evidence scope. Quota acceptance, sampled intent and task lifetime are useful quantities when named precisely; none should acquire a stronger meaning merely because it is exported as CSV.

Chapter 29 – Supply Chain Designs Without False Guarantees

Atlas API can now carry a digest through service intent, but an attacker could provide a digest for a malicious image. Precise identity is a prerequisite for a useful trust decision, not the decision itself. This chapter studies how the repository proposes to add signature and vulnerability evidence while keeping those proposals separate from the running application.

You already know the parser-only default resolver and built-in admission rules. Here you will define the artifact subject, place verification at explicit boundaries, and evaluate stale or unavailable evidence without pretending a scanner ran. The Atlas lab is a small executable book-side decision contract using supplied fake verdicts. It contacts no registry, verifies no signature and scans no image.

Digest Identity and Signature Authenticity

A digest addresses content identity: the decision can name the same selected artifact across review, rollout and execution. A signature decision asks whether an accepted identity or key authorized that artifact under a particular trust policy. Even a valid trusted signature does not prove absence of vulnerabilities or malicious behavior. These are separate questions with different evidence and failure modes.

The source's implemented image policy checks registry, digest presence and mutable/latest rules. The default ParserResolver parses metadata without remote resolution. No signature verifier is called by that parser or the ordinary built-in admission evaluator. The signing document explicitly describes a design, including its goals and non-goals. Claim ch29.baseline.

docs/IMAGE_SIGNING_DESIGN.md, lines 7–7 at the locked revision.

> This design does not implement signing, run a certificate authority, mirror registries, scan images for vulnerabilities, or make a signature prove that the image is safe. SBOM and vulnerability decisions are covered by `SUPPLY_CHAIN_SECURITY_DESIGN.md`.

For Atlas, define the threat before choosing a mechanism. The design addresses substituted tags, unauthorized publishers, stale results and artifact substitution between admission and execution. It does not protect a host already compromised at root, a deliberately malicious trust administrator or a compromised build identity that still satisfies the configured policy. Naming these exclusions prevents a signature badge from becoming a claim of universal safety.

A useful artifact subject contains the immutable digest plus relevant repository/platform interpretation. The trust decision also needs namespace and policy revision. Otherwise a result accepted for a development namespace could be replayed for production, or a success under an old policy could be reused after trust roots changed. The proposed verification evidence should bind those dimensions explicitly rather than storing an unqualified boolean verified.

Signed authenticity and registry access remain distinct. A registry credential permits retrieval; it should not become a signing credential or appear in a verification record. Likewise, the private signing key should not be sent to Orch so that it can verify signatures. The design describes public trust material and identity-based checks; actual key management and adapter behavior require a separate implementation and review.

The supplied-verdict lab intentionally avoids realistic-looking image digests. Its subject string says synthetic-non-executing-digest, because it will never be passed to an Engine pull. This makes the scope visible in both code and output. A future live verification exercise must use an available artifact and independently checked tool/version documentation rather than turning the teaching string into an operational reference.

Trust Policy and Verification Placement

The signing design proposes verification at API admission, rollout creation and the agent before execution. Each point answers a different failure window. Admission gives early feedback, rollout ties a version to its exact subject, and the agent can refuse a stale or bypassed assignment at the final execution boundary. These are proposed enforcement points, not active hooks demonstrated by the current baseline. Claim ch29.placement.

An admission-only success can become stale before a delayed task starts. A rollout-only success does not prevent a direct agent path from bypassing that controller. An agent-only rejection gives late feedback after the control plane already accepted intent. Multiple checks can narrow those windows, but only if they bind the same digest and policy context and have a tested protocol for expiry and unavailable dependencies.

The design suggests short-lived assertions for agents. A complete assertion must bind subject, namespace, policy revision, expiry and issuer, and the agent must authenticate that binding. Copying a JSON object containing allowed:true is not equivalent. The book-side Verdict is deliberately just supplied test input, so its boolean signature_trusted demonstrates decision wiring only; it is not a cryptographic assertion implementation.

Trust-root rotation also changes the decision. If a previously accepted root is removed, cached results under the old policy cannot automatically remain valid forever. Record the policy revision and define how running and pending workloads are reevaluated. A grace period may be operationally necessary, but it needs an explicit scope and evidence rather than a silent exception.

Place audit evidence at the decision boundary. Useful records include digest, namespace, policy revision, verifier version, result reason and expiry. They should exclude registry passwords, raw identity tokens and

full sensitive scanner payloads. A source event such as verification.succeeded is a proposed schema here; its name in a document does not mean the default server currently emits it.

The acceptance path should distinguish invalid evidence from unavailable evidence. An untrusted signature, wrong subject or expired assertion is not merely a transient network outage. Retrying those as though another HTTP attempt would make them valid can hide a policy error. Conversely, an unavailable verifier should not be mislabeled as proof of a malicious publisher. Stable reason codes support different operational responses.

SBOM, Vulnerability Freshness, and Fail Modes

The supply-chain design proposes SBOM records and scan results keyed by immutable digest. It separates large documents from metadata, retaining document identity, producer, scanner/database version and timestamps. This is design material: the proposed image_artifacts, image_sboms and vulnerability_scans records are not evidence that the current migrations/API implement that data plane. Claim ch29.scanning.

docs/SUPPLY_CHAIN_SECURITY_DESIGN.md, lines 34–34 at the locked revision.

```
Admission runs after tag-to-digest resolution. Missing/stale data,
active scans, scanner outage, and policy violation are separate
structured results. Production defaults to fail closed for missing
or expired evidence. Development may allow deployment while a scan
is pending, but this produces an audit event.
```

An SBOM describes a claimed component inventory. Its existence is not proof that every component is accurately identified or that no vulnerability exists. A scanner verdict depends on the artifact, scanner behavior and the vulnerability database available at scan time. New information can make yesterday's clean result insufficient without any change to image bytes. Evidence freshness therefore belongs in policy rather than being inferred from an immutable digest.

A meaningful scan decision needs at least the subject, trusted producer/scanner, database version, completion state, scan time and expiry. Missing, pending, failed and expired results are different states. A critical finding is another different state. The proposed policy can choose how to treat each, but should not collapse all of them into a generic passed flag.

For Atlas production, the illustrative contract denies missing/unavailable or expired evidence and any critical finding. A narrowly selected development option allows a pending scan only after subject binding, signature trust and signature expiry have passed. That option returns allow_with_audit rather than an unqualified success. It does not override an explicitly untrusted signature or a mismatched namespace. Claim ch29.contract.

The contract's name matters: returning allow_with_audit does not actually persist an audit record. A future caller would need to perform that obligation and define what happens when the append fails. Similarly, signature_trusted is supplied by a fake, not computed by a real verifier. The executable test is useful precisely because it isolates the decision truth table without inventing external integration evidence.

Supplied condition	Teaching decision	Scope
Matching subject/policy, fresh trusted verdict, no critical finding	Allow	Supplied inputs only
Another namespace, digest or policy revision	Deny subject mismatch	Prevents modeled replay
Untrusted or expired signature verdict	Deny	No real signature verification
Pending scan with explicit narrow exception	Allow with audit obligation	Caller must implement audit
Unavailable scanner or expired completed result	Deny	Illustrative outage policy

The decision table has a deliberate order. Subject mismatch and signature failure are checked before the pending-scan exception. A broad except-on-error rule placed first could accidentally bypass trust checks. The tests vary namespace, digest and policy separately, then exercise trust, expiry, findings and the narrow exception. The actual teaching run passed all five test methods and their subcases.

This observed success means the small Python contract behaves as asserted with injected time and fake verdicts. It does not establish real signature validity, SBOM parsing, scanner correctness, authenticated cache storage or a durable audit trail. Those adapters remain proposed. Keeping that limit explicit makes the test reusable when a real adapter is eventually introduced: its output must conform without changing the decision's meaning.

What the Repository Proposes Rather Than Runs

The design includes proposed supply-chain API routes and CLI commands. They must not be copied into a current command reference as though implemented. A reader trying an absent scan command would be misled even if the architecture is sensible. The correct baseline description is implemented digest/reference policy plus separate signing/scanning designs.

History reinforces that boundary. f2836eabfff8e442d38984433f3e1f5c642667d1 adds the relevant image pinning and design material in a batch with other work. Its actual changed paths do not demonstrate a deployed verifier, SBOM service or vulnerability scanner. Original prompts 68 and 69 request design, and their existence should not be promoted to execution evidence.

A future integration plan should begin with versioned contracts and threat-model acceptance criteria. Pin the verifier/scanner version, verify commands against official documentation, and record a real artifact subject and expected rejection cases. Do not install a moving latest tool

and then describe its output as historically representative of this source revision. The book does not need external installation to evaluate its current design boundary.

Cache design needs particular attention. Key results by subject, namespace or applicable trust scope, policy revision and verifier/database version, with explicit expiry. Authenticate stored results and avoid extending expiry simply because a cache entry was read. A fresh scan with the wrong policy context is not reusable evidence, and a cache outage should follow the same explicit fail mode as an unavailable backend.

Decide separately what happens to a running image when new findings arrive. Immediate termination can reduce availability without producing a safe replacement. Alert-only handling can preserve service while leaving known risk active. The design proposes a staged approach and future policies; the baseline does not implement an automatic quarantine guarantee. A rollout-blocking rule and a running-workload eviction rule require different tests and operator expectations.

For the safe Atlas lab, start in the book's teaching directory and run the deterministic tests. Python's standard library is sufficient. No Docker, registry, network, signing key, scanner database or application process is required.

```
cd books/inside-orch/examples/teaching-extensions/ch29
python3 -B test_contract.py
```

The clock is a fixed UTC instant. First predict each table outcome, including equality at expiry, which is denied. Then run and inspect the recorded output. Cleanup is process exit; -B avoids bytecode artifacts. A failed assertion should be investigated as a contract bug rather than weakening the expected denial to match an accidental result.

For a design extension, add a policy-revision change while a task waits to start, and decide whether its old supplied verdict remains acceptable. Keep that change visibly book-side and test it independently. A real adapter would additionally prove how it obtains and authenticates

evidence. Until that exists, the correct release language is a tested illustrative decision contract accompanying an unimplemented supply-chain architecture.

Exercises

1. Explain how a digest can identify malicious content precisely without proving a trusted publisher authorized it.
2. Identify the failure window addressed by admission, rollout and agent verification separately. Which are active in the baseline?
3. Why must a completed scan bind database version and expiry? Explain how new findings can change acceptance without changing the image.
4. A developer moves the pending-scan exception before subject/signature checks. Which modeled bypass becomes possible, and which test should catch it?

Worked answers preserve the distinction between identity, trust and freshness. The book's concrete achievement here is a reviewed design and an executed supplied-verdict contract, with every external adapter and stronger operational guarantee still labeled as future work.

Chapter 30 – GitOps, Drift, and Controlled Reconciliation

Atlas API intent now lives in a Git repository. An operator can point to a committed manifest and ask which revision produced the service. That improves traceability, but it does not automatically make a repository apply atomic or every deletion safe. The controller still reads files, calls mutations and records progress across several failure windows.

This chapter assumes shared service parsing, admission and namespace scope. You will follow an actual local Git checkout into MemoryService, compare warn with auto_revert after manual scaling, and inspect the exact prune predicate. The safe lab uses real temporary commits and no containers. Its observations establish control-plane behavior for that fixture rather than a production deployment pipeline.

A Source Revision You Can Trace

GitOpsSource identifies a repository URL, branch, path, sync interval, namespace, drift policy and prune setting. ValidateSource requires a nonempty repository/branch, positive interval and a supported drift policy. It rejects URL user information and credential-shaped query parameters. A relative configured path must remain inside the repository. Claim ch30.source.

CommandFetcher creates an owned temporary directory and invokes git clone with an argument slice, then resolves HEAD through rev-parse. The returned Checkout contains Directory, Revision and Cleanup. Sync defers that cleanup after a successful checkout. This gives the controller a concrete revision and a bounded checkout lifetime without mutating the pinned source or keeping an unmanaged checkout.

internal/gitops/gitops.go, lines 73–85 at the locked revision.

```
        cleanup := func() { _ = os.RemoveAll(directory) }
        args := []string{"clone", "--depth=1", "--single-branch",
"--branch", source.Branch, "--", source.RepositoryURL, directory}
        command := exec.CommandContext(ctx, "git", args...)
        if output, err := command.CombinedOutput(); err != nil {
                cleanup()
                return Checkout{}, fmt.Errorf("clone repository: %w:
%s", err, strings.TrimSpace(string(output)))
        }
        revisionBytes, err := exec.CommandContext(ctx, "git", "-C",
directory, "rev-parse", "HEAD").Output()
        if err != nil {
                cleanup()
                return Checkout{}, fmt.Errorf("resolve checkout
revision: %w", err)
        }
        return Checkout{Directory: directory, Revision:
strings.TrimSpace(string(revisionBytes)), Cleanup: cleanup}, nil
```

A branch name is still mutable. The important evidence is the checked-out commit, not an assumption that main always means the same content. Local clone depth behavior can differ from a remote transport, but the resolved HEAD identifies the actual fixture revision. Record that value in the lab rather than inventing a short placeholder commit or treating a previous source revision as the new checkout.

The canonical teaching fixture creates a temporary repository, commits atlas.yaml, synchronizes it and later commits its removal. Its actual execution record contains both full manifest-repository commit IDs in stdout. Those IDs identify this disposable content history; they are distinct from the pinned Orch source SHA used to execute the controller. The manifest commits belong only to the disposable fixture.

Credentials in repository URLs are dangerous because failure messages can include command output and source metadata. The implementation rejects obvious URL-userinfo and credential query forms, but that is not a promise to sanitize every possible secret-bearing remote string or Git

configuration. The lab therefore uses a local path and no credentials. A real private-source integration needs a separate reviewed authentication mechanism.

Fetch, Parse, Admission, and Apply

After checkout, manifestFiles selects the configured YAML file or walks a configured directory, collects .yaml/.yml paths and sorts them. Deterministic path order makes partial outcomes reproducible. It does not make all files one transaction. Sync reads, parses and applies each file inside the loop before moving to the next. Claim ch30.apply.

internal/gitops/gitops.go, lines 218–235 at the locked revision.

```
        syncCtx := namespace.WithContext(ctx, source.Namespace)
        managed := make([]string, 0, len(files))
        for _, file := range files {
                data, err := os.ReadFile(file)
                if err != nil {
                        return controller.fail(syncCtx, source,
fmt.Errorf("read manifest %q: %w", file, err))
                }
                spec, err := controller.parser(data)
                if err != nil {
                        return controller.fail(syncCtx, source,
fmt.Errorf("validate manifest %q: %w", file, err))
                }
                service, err :=
controller.store.ApplyService(syncCtx, spec)
                if err != nil {
                        return controller.fail(syncCtx, source,
fmt.Errorf("apply service %q: %w", spec.Name, err))
                }
                if marker, ok := controller.store.(managedStore); ok
{
                        relative, _ :=
filepath.Rel(checkout.Directory, file)
                        policy := source.DriftPolicy
```

The server supplies cli.ParseDeploy, and ApplyService enters the checked service create/update paths. This is the ordinary shared parsing/admission boundary discussed earlier. Namespace context comes from

the GitOps source, so manifests are applied within that selected scope. Successful services can be marked with source ID, checkout commit, relative path, desired spec, policy and last-check time.

Suppose atlas-a.yaml is valid and applied, while atlas-b.yaml contains an unknown field. The controller fails on the second file and records source.LastError, but the first service mutation has already happened. There is no repository-wide rollback in this loop. The incident record must inspect live services as well as the source status; a failed sync does not imply nothing changed.

Source.LastRevision is updated after the successful loop and optional prune. A failed sync can therefore leave prior revision metadata while an earlier file from the new checkout has already changed a service. This is an important provenance limit. A future all-or-nothing apply would need a deliberate validation/planning and transaction/operation protocol rather than merely changing the error message.

The failure helper records the error and sync-attempt time, then emits failure observations when source update succeeds. Event and audit appends in the emitter are best effort. A successful source update is not a durable mutation journal of every per-file action, especially with default MemoryService. Retain actual fixture output and service state before cleanup instead of relying on an assumed complete history.

Repeated application of an unchanged normalized spec should avoid creating unnecessary versions. The upstream local-repository test explicitly synchronizes twice and asserts the version does not advance. The focused executed suite passed that case alongside local prune and credential-URL rejection. It uses real Git commands and in-process state, with no external repository or Docker deployment.

Warn Versus Auto-Revert Drift

MemoryService stores a GitOpsManagedState with DesiredSpec. GitOpsStatus compares current service.Spec against that stored desired value using DeepEqual. It marks missing sources unknown, stored errors

sync_error, equality in_sync and differences drifted. The exact comparison is the code's normalized stored representation, not a semantic diff of every physical container. Claim ch30.drift.

With warn, a manual scale can remain live while status reports drifted and emits a drift observation. With auto_revert, that same status method restores DesiredSpec, increments deployment version, updates the timestamp and calls immediate service reconciliation. It then marks the service in_sync. This is a mutating status path, not merely a passive read or an independently proven continuously running drift controller.

The Atlas fixture begins with replicas zero in the committed manifest. It manually scales to two, checks warn and verifies two remain. It then changes the stored managed policy to auto_revert, manually scales to three and calls status. The returned service has replicas zero and in_sync. These values are actual assertions in the recorded run. No workload processes existed, so the result describes accepted service intent only.

The restoration branch does not call admission or quota checking again in the inspected method. A stored desired spec that was accepted earlier can therefore be restored through a different path from ordinary ApplyService. This matters if policy or quota changed since the original acceptance. Describe the exception openly instead of promising every state restoration reruns current acceptance rules.

Operation	Current behavior	Boundary
Ordinary source apply	Shared parser and service mutation	Admission on that path
Status with warn	Detects difference and retains live spec	Observation does not repair
Status with auto_revert	Restores stored desired spec and reconciles	Mutating status; no fresh Admit call there
Container manually changed	Separate agent/runtime drift question	Spec equality is not physical equivalence

A manual rollback or emergency scale can conflict with auto_revert. If an operator changes live state and another process reads status, the status call can restore the old desired spec. Operational procedures should specify which source/policy is authoritative during intervention. A UI labeled read-only status should not obscure this behavior. A future design can separate detection from repair and require checked repair operations, but that is not the baseline method.

The failure window is between storing desired intent and later restoring it under changed constraints. A focused teaching test could tighten quota after scaling down, then ask status to restore a larger stored desired count. It should record whether current checks are re-run, not silently alter the baseline. This exposes an integration question that happy-path equality tests cannot answer.

Prune Only What This Source Owns

The current prune loop constructs wanted names from the new manifest set and previous names from source.ManagedServices. It lists services in the source namespace and deletes a service when its name was previously managed and is no longer wanted. It preserves unrelated names. That is the exact implemented predicate; it does not compare the current service's ID or GitOps.SourceID before deletion. Claim ch30.prune.

The lab's unrelated service is intentionally never in the prior managed-name set. Removing atlas.yaml in a new real commit makes Atlas deleted, while unrelated stays active. This passed. It establishes the tested name-set boundary, not an unconditional guarantee that every currently unrelated object is immune to prune.

Consider replacement or ownership transfer under a previously managed name. Another source can mark the same service managed, or an operator can create a replacement after old state is removed under a different lifecycle. If its name still belongs to the first source's stale ManagedServices, that source's prune predicate does not independently check the new ownership marker. Names alone are weaker than stable object/source identity.

A safer teaching design would require namespace, current object ID and current source ownership to agree before deleting, with a conflict outcome when they do not. It would also define explicit transfer semantics and update managed inventories atomically enough for its intended guarantee. That guard is a proposal here, not a silent patch to the immutable controller. The real fixture deliberately avoids ownership replacement and does not claim to have tested that stronger predicate.

Prune also changes desired service lifecycle rather than directly sweeping daemon resources. Agent cleanup follows its own task ownership rules. A source removal should not trigger broad Docker container or volume pruning. For stateful workloads, data retention must be an explicit decision independent of whether a manifest disappeared. Chapter 32 examines why workload deletion and persistent data deletion are different operations.

Development history spans two batches. f2836eabfff8e442d38984433f3e1f5c642667d1 introduces the GitOps source/checkout path with other work. 4d641bbe8913da568d6d9f4697a0144314e10b87 adds later drift behavior. These actual changes are more precise than attributing every capability to the original GitOps prompt alone.

For the safe Atlas lab, copy examples/teaching-extensions/ch30/main.go into book-gitops inside the disposable source module. Use the pinned Go toolchain and Git; no database, Docker daemon, server listener or external account is required. The fixture owns its temporary manifest repository and lets CommandFetcher own each temporary checkout.

```
go run ./book-gitops
go test ./internal/gitops ./internal/controlplane \
  -run 'TestControllerSyncsLocalFixtureRepositoryAndPrunes|
TestSourceRejectsPlaintextCredentials|
TestGitOpsManualChangesCreateDriftAndAutoRevert' \
  -count=1 -v
```

Predict both policy outcomes and the preserved unrelated name before execution. Success requires real full commit IDs, warn retaining two replicas, auto_revert restoring zero, Atlas deleted after the removal commit and unrelated active. The recorded teaching run passed those assertions. Cleanup removes only the fixture's exact temporary repository; checkout Cleanup handles its own directory. No external repository is modified.

Exercises

1. Why must a lab retain both the Orch source SHA and the manifest repository's checked-out commit?
2. The second sorted manifest fails parsing after the first applied. What can LastError and LastRevision tell you, and what must be inspected separately?
3. Explain why GitOpsStatus under auto_revert is not a passive observation and why its acceptance path differs from ApplyService.
4. Design an ownership-transfer prune test. Which extra predicate is absent from the baseline, and what does the passing unrelated-name fixture actually prove?

Worked answers keep provenance, drift and deletion scope distinct. Traceable Git intent is valuable when partial apply, mutating status and name-based pruning remain visible rather than being hidden under a blanket GitOps reliability claim.

Chapter 31 – Finite Jobs and Time-Based Work

Atlas needs a report that exits after counting a bounded input. A service controller would interpret exit as lost replica capacity, while a Job needs to interpret it as an outcome. Scheduling the same report every morning introduces another problem: local clock meaning, overlap and delayed controller iterations. These concerns require identities and state transitions beyond a long-running service's replica count.

This chapter assumes task reports, pending assignment and namespaces. You will follow one successful attempt and one retry, calculate five-field schedule occurrences with injected time, and distinguish Allow, Forbid and Replace. The safe lab executes real control-plane and schedule code with fake reports/clocks; it does not claim the finite workload ran on Docker.

Finite Attempts and Exit Codes

JobSpec contains name, image, command, backoffLimit, resources, secrets, placement and volumeClaims with the source's exact YAML names. It is not a Kubernetes Job wrapper. CreateJob validates its spec, supplies resource defaults, checks the namespace and rejects an already-active same-named Job there. It creates a Job with Attempts one and one pending task carrying JobID, command and resource/placement data. Claim ch31.attempt.

The task is pending rather than directly assigned through the ordinary service path. Scheduler.Plan can construct a service-like scheduling view for Job tasks from their task fields, and MemoryService.AssignTask records the chosen node with its optimistic guard. This makes the earlier scheduler discussion relevant without assuming service and Job creation share every validation or credential feature.

Once a report arrives, updateJobForTaskLocked maps running/healthy to JobRunning. A stopped task with exit code zero produces JobSucceeded and CompletedAt. A stopped nonzero exit or failed task enters retry-or-fail handling. A missing stopped exit code defaults to zero at this method; a missing failed exit code defaults to one. This is the exact accepted-report interpretation, not proof that an arbitrary missing code is physically trustworthy.

internal/controlplane/workloads.go, lines 190–216 at the locked revision.

```
	switch task.ActualStatus {
	case types.TaskRunning, types.TaskHealthy:
		job.Status = types.JobRunning
	case types.TaskStopped:
		code := 0
		if task.ExitCode != nil {
			code = *task.ExitCode
		}
		job.LastExitCode = &code
		if code == 0 {
			job.Status = types.JobSucceeded
			job.CompletedAt = now
		} else {
			s.retryOrFailJobLocked(&job, task, now)
		}
	case types.TaskFailed:
		code := 1
		if task.ExitCode != nil {
			code = *task.ExitCode
		}
		job.LastExitCode = &code
		s.retryOrFailJobLocked(&job, task, now)
	}
	job.UpdatedAt = now
	s.jobs[job.ID] = job
}
```

The actual focused Job test assigns a task, injects running and stopped-zero reports, and checks succeeded. It then creates a failing Job and injects nonzero failures through two attempts. The recorded suite passed.

Because the reports are supplied directly and no Engine exists in this fixture, describe it as state-transition verification rather than successful execution of the report command.

A Docker execution experiment needs additional evidence: available image, actual command, container exit code, task identity and Job outcome. Command fields do reach the agent's ContainerSpec, but private service pull credentials and environment resolution do not automatically apply to the separate Job branch. A finite workload feature should be tested through its own DTO and agent path before claiming parity with services.

Retry Limits and Terminal Outcomes

retryOrFailJobLocked compares current Attempts with BackoffLimit. With limit zero, the first failure is terminal because Attempts starts at one. With limit one, the first failure creates a second attempt, and the second failure is terminal. Thus the value bounds retries beyond the initial attempt. The helper does not implement a time-delay backoff merely because the field name contains backoff. Claim ch31.retry.

internal/controlplane/workloads.go, lines 217–241 at the locked revision.

```
func (s *MemoryService) retryOrFailJobLocked(job *types.Job,
previous types.Task, now time.Time) {
        if job.Attempts > job.Spec.BackoffLimit {
                job.Status = types.JobFailed
                job.CompletedAt = now
                return
        }
        job.Attempts++
        task := previous
        task.ID = types.TaskID(newUUID())
        task.NodeID = ""
        task.ContainerID = ""
        task.ActualStatus = types.TaskPending
        task.DesiredStatus = types.TaskRunning
        task.RestartCount = job.Attempts - 1
        task.ExitCode = nil
        task.FailureReason = ""
        task.CreatedAt = now
        task.UpdatedAt = now
        task.StartedAt = time.Time{}
        task.FinishedAt = time.Time{}
        s.tasks[task.ID] = task
        job.TaskIDs = append(job.TaskIDs, task.ID)
        job.Status = types.JobPending
}
```

A retry copies the previous task but assigns a new task ID, clears node/container identity, returns it to pending/running intent, resets exit/failure/timing fields and appends the new ID to Job.TaskIDs. RestartCount records attempt-related retry count. Preserving the prior task lets an operator distinguish attempts instead of overwriting one task's history as though a single process simply changed its past outcome.

The finite work itself must tolerate duplicate effects. A task can write a report externally and then lose its success acknowledgement. A later attempt may repeat that report. The Job controller's terminal-state logic does not make the external write exactly once. Use an application-level idempotency key or transactional output protocol when duplicate side effects would be harmful, and test that independently from the orchestration status.

A further source boundary concerns adoption after agent reconstruction. The create label uses a job-prefixed workload identity, while the shared matching helper compares a service-ID label against task.ServiceID. Job tasks have no ordinary service ID. That mismatch is a source-derived integration concern, not a tested guarantee of correct Job adoption. Keep it in the evidence ledger instead of borrowing the service adoption result.

DeleteJob removes Job metadata and marks its associated tasks for removal, detaching recorded task volumes in that path. Physical container stop/removal still belongs to agent convergence. Deleting the control-plane record is not proof every host side effect has ended. For a live finite-job lab, collect the actual exit and task history before cleanup, and verify only run-owned containers are removed.

Cron Expressions and Explicit Timezones

CronJobs have schedule, timezone, concurrencyPolicy, jobTemplate, suspended and missedScheduleLimit. CreateCronJob defaults timezone to UTC and concurrency to Allow, validates supported policy values and the Job template, then stores the item. It does not parse the cron expression or load the timezone in that creation method. Those checks occur in Controller.RunOnce, so some invalid schedules/timezones fail later during controller processing.

The parser requires exactly five fields: minute, hour, day of month, month and weekday. Bounds are0–59,0–23,1–31,1–12 and 0–6. It supports comma lists, ranges, wildcard and positive step syntax implemented by parseField. Do not add a seconds field, named weekdays/months or a seventh-field extension from another cron implementation. Claim ch31.schedule.

matches requires all five fields to match. In particular, day-of-month and weekday are combined by this all-field rule; readers should not import another cron dialect's alternative-day semantics. Next truncates to a

minute, starts at the following minute and searches a bounded year-sized number of minute increments. It can return zero if no occurrence is found within that horizon.

The current CronJob CLI accepts a smaller manifest surface than the full API type. A supported illustrative CLI manifest is:

```
name: atlas-report-schedule
schedule: "0 9 * * 1-5"
timezone: UTC
concurrencyPolicy: Forbid
jobTemplate:
  image: busybox:1.36
  command: ["sh", "-c", "printf 'atlas report\\n'"]
  resources:
    cpu: 100m
    memory: 1Mi
```

This shows the actual intermediate CLI envelope and quantity strings; it is not an executed Docker manifest or proof the image is available. The CLI constructs a CronJobSpec from those fields, setting template name from the schedule name and identical resource requests/limits. It does not forward missedScheduleLimit or template backoffLimit from YAML in that envelope. The API/domain type supports additional fields, so use its explicit JSON contract when those are needed. The one-shot Job run envelope does support its own backoffLimit. Unknown YAML fields can be ignored by these simple unmarshalling paths; never infer acceptance of a field from a command returning without a parse error.

RunOnce loads the explicit IANA location and evaluates the schedule in that location, storing schedule timestamps in UTC. Nine in the morning under UTC and nine under Europe/Belgrade are different instants on the fixture date. The book's actual arithmetic fixture observed09:00UTC for UTC and 07:00UTC for Belgrade on2026-10-13, plus rejection of six fields. It uses injected dates, not wall-clock waiting.

Timezone arithmetic alone does not settle every daylight-saving policy. Minute stepping and local-time matching can encounter absent or repeated wall-clock times. The exact date/location must be part of a DST

test; do not infer a universal skipped/repeated-hour guarantee from the ordinary October13 case. The dedicated workload lab adds explicit transition cases with recorded outcomes.

Allow, Forbid, Replace, and Missed Runs

RunOnce chooses its base from LastScheduleAt, or CreatedAt if none exists. Between returns occurrences after that base through now, up to the configured missed limit, defaulting to100 when nonpositive. It takes the earliest due occurrences in that window. A delayed iteration can therefore leave a backlog rather than silently selecting only the latest minute. Claim ch31.concurrency.

Active jobs are those referenced by ActiveJobIDs whose current status is pending or running. Allow retains them and creates due jobs. Forbid makes due nil when an active job exists. Replace deletes active jobs before creating the due set. Replace requests removal; it does not establish that every old host process stopped before new work starts. This matters when old and new attempts write to the same external resource.

Policy	Existing active Job	Controller action	Physical guarantee absent
Allow	Retained	Creates due Jobs	Unique external effect
Forbid	Retained	Suppresses this pass's creation	Permanent skipping of backlog
Replace	Deleted through store method	Creates due Jobs afterward	Verified old-process fencing

Forbid does not advance LastScheduleAt when it suppresses all due occurrences in the inspected loop. Later, after the active job finishes, those earlier occurrences can still be due, subject to the limit. Also, when no active job exists at the start, a backlog's several due entries can be created in one Forbid pass; the controller does not recalculate overlap between each newly created Job in that loop. Name the tested behavior rather than importing a stronger non-overlap promise.

Job names include the UTC occurrence timestamp. That provides a useful deterministic label, but CreateJob only rejects same-name Jobs while active. Consider a failure after CreateJob succeeds but before UpdateCronJob records LastScheduleAt. The next iteration can revisit that occurrence. An active duplicate may reject, while later lifecycle changes can permit reuse. There is no transactional create-and-schedule checkpoint or exactly-once external execution guarantee in this method.

The default server constructs the Cron controller, so this is not wholly disconnected design. Its authoritative state is still MemoryService, and fresh process state loss can remove schedule/job inventory. A durable scheduling claim needs persistent composition, atomic checkpoint semantics and restart/partition tests beyond the current fake-clock suite.

History is the batch 4d641bbe8913da568d6d9f4697a0144314e10b87, which adds jobs, cron, local storage, DNS and notifications. The design includes broader stateful/time-based ideas; the code and executed tests define which pieces are present. Do not invent a separate implementation commit for each original workload prompt.

For the safe lab, use the disposable module and Go 1.25. Run the focused tests, then copy examples/teaching-extensions/ch31/main.go to book-cron and run the arithmetic fixture. No Docker, database, server listener or external service is required.

```
go test ./internal/cronjobs ./internal/controlplane \
  -run 'TestParseAndRunWithFakeClock|
TestConcurrencyForbidAndReplace|TestJobSuccessFailureRetryAndDelete'
\
  -count=1 -v
go run ./book-cron
```

Expected observations are one successful injected attempt, bounded retry terminal failure, two earliest missed minutes under a limit of two, and distinct explicit-timezone instants. The linked runs passed those assertions. Cleanup is process exit and optional removal of copied teaching directories; no persistent report data was produced.

A useful schedule incident bundle includes the original five-field expression, IANA location, UTC base/now, computed due list, active Job IDs and the last accepted checkpoint. Preserve these before deleting generated work. A local-time string without its location or UTC offset cannot explain a repeated-hour decision, and a Job count alone cannot reveal whether the controller replayed backlog or generated a new occurrence.

Exercises

1. Calculate maximum total attempts for backoffLimit zero and one. Why does the helper not prove delayed retry?
2. Explain why an accepted stopped report without an exit code and a failed report without an exit code take different defaults.
3. For a three-minute delay and missed limit two, which occurrences are created first? What can Forbid do when several are due and none was active at the start?
4. Identify the create-before-checkpoint failure window and design evidence needed for a stronger scheduling guarantee, including a concrete DST transition case.

Worked answers retain attempt identity, exact cron syntax and time scope. Finite outcomes and deterministic scheduling are implemented, while external exactly-once effects, physical replacement fencing and durable process recovery remain separate requirements.

Chapter 32 – Local Volumes, Claims, and Workload Topology

Atlas API needs a disposable cache that survives a container replacement on one worker. A named local volume can preserve those bytes, but it also changes placement: moving the task to another worker does not move its data. The orchestration model must distinguish a logical claim, a physical local volume and the current authority to write it.

This chapter assumes task assignment, agent mounts and namespace scope. You will trace claim resolution into an attachment, explain the implemented ReadWriteOnce predicate, and identify which topology checks reach ordinary service placement. The safe lab uses control-plane and scheduler tests with disposable metadata; no real database contents or host volume are touched.

Local Volumes and Claim Binding

Volume contains ID, namespace, name, driver and optional node ID. CreateVolume defaults the driver to local and rejects another driver. Names are unique within a namespace in the MemoryService inventory. This describes local volume metadata; it is not a distributed storage provisioner or an actual Engine volume-create call in that method. Claim ch32.binding.

VolumeClaim adds its own identity/name, optional VolumeID, access mode and allow_concurrent_writers. CreateVolumeClaim defaults access mode to ReadWriteOnce and checks that an explicitly referenced volume exists in the same namespace. It stores the claim under a namespaced name key. The current method replaces that map entry rather than implementing a general immutable-claim lifecycle with a duplicate-name rejection; do not assume semantics from another platform.

resolveVolumeMountsLocked takes the workload namespace and each claim reference. If the claim has no bound VolumeID, it searches for a same-name local metadata volume or creates one. It then produces ResolvedVolumeMount with claim/volume IDs, volume name, current node binding, target, read-only flag, access mode and concurrent-writer option. These resolved fields become part of the task snapshot.

internal/controlplane/workloads.go, lines 460–477 at the locked revision.

```
                if c.VolumeID == "" {
                        for id, v := range s.volumes {
                                if v.Namespace == ns && v.Name ==
c.Name {
                                        c.VolumeID = id
                                        break
                                }
                        }
                        if c.VolumeID == "" {
                                v := types.Volume{ID: newUUID(),
Namespace: ns, Name: c.Name, Driver: "local", CreatedAt: s.now()}
                                s.volumes[v.ID] = v
                                c.VolumeID = v.ID
                                s.volumeClaims[namespacedKey(ns,
c.Name)] = c
                        }
                }
                v := s.volumes[c.VolumeID]
                out = append(out, types.ResolvedVolumeMount{ClaimID:
c.ID, VolumeID: v.ID, VolumeName: v.Name, NodeID: v.NodeID, Target:
m.Target, ReadOnly: m.ReadOnly, AccessMode: c.AccessMode,
AllowConcurrentWriters: c.AllowConcurrentWriters})
        }
        return out, nil
```

The distinction between claim identity and volume identity helps describe retention. Removing one task need not delete the claim or volume metadata. A later task can resolve the same claim and find the same volume. That is useful for local replacement, provided its topology and physical naming remain correct. It does not guarantee data backup or cross-host access.

The agent converts a resolved mount into the runtime VolumeMount using VolumeName, Target and ReadOnly. The Engine adapter builds name:target:mode bind strings. It does not include the metadata volume UUID or namespace in that physical name mapping. Two logical namespaces with the same volume name on one daemon therefore require particular scrutiny: the metadata separation alone does not prove separate physical volume names. Claim ch32.physical-name.

For Atlas, choose a unique run-owned physical volume name in any live experiment. A friendly generic data name can collide with an existing host volume even if the control-plane record is new. Inspect the actual mount name and container before writing disposable data. The source-only fixture avoids this risk and cannot establish physical namespace isolation.

Attachments and Active Writer Rules

VolumeAttachment records volume ID, task ID, node ID, read-only mode and attach/detach timestamps. attachTaskVolumesLocked creates an attachment for each resolved mount and binds an unbound volume's metadata NodeID to the chosen node. detachTaskVolumesLocked marks matching active records detached instead of erasing their history. Claim ch32.attachments.

The eligibility predicate first excludes a node when a resolved mount has a different nonempty NodeID. For a writable ReadWriteOnce mount without allow_concurrent_writers, it then excludes placement when any active attachment exists for that volume. That is the actual predicate, including its conservative treatment of existing attachments. Do not replace it in prose with a more permissive abstract interpretation.

internal/controlplane/workloads.go, lines 492–507 at the locked revision.

```
                for _, m := range mounts {
                        if m.NodeID != "" && m.NodeID != n.ID {
                                ok = false
                                break
                        }
                        if !m.ReadOnly && m.AccessMode ==
types.VolumeReadWriteOnce && !m.AllowConcurrentWriters {
                                for _, a := range s.attachments {
                                        if a.VolumeID == m.VolumeID
&& a.DetachedAt.IsZero() {
                                                ok = false
                                                break
                                        }
                                }
                        }
                }
                if ok {
                        out = append(out, n)
```

A new read-only mount skips the writable-RWO conflict branch, so it does not request the writer exclusion there. However, the loop for a later writable request checks active attachments without filtering their ReadOnly flag. An existing read-only attachment can therefore block that later writer in this implementation. The storage documentation's simple statement that readers do not consume a writer lock needs this code-level qualification.

allow_concurrent_writers bypasses that writer-conflict branch; it does not add application-level serialization. ReadWriteMany likewise does not create distributed storage just because the access mode name exists. The selected Docker driver remains local. Multiple allowed writers can corrupt application data unless the application and filesystem usage are designed for it.

An attachment is a control-plane record, not a kernel lease. Marking it detached cannot prove an unreachable worker's process stopped writing. Conversely, a missing attachment after memory state loss does not mean

the physical volume is unused. Logical writer exclusion is valuable within the authoritative inventory, but distributed fencing requires an external mechanism that actually revokes stale write authority.

The removal lifecycle also matters. Reported task removal and Job deletion invoke detach logic in their relevant paths. DeleteVolume rejects a metadata volume with active attachments. This prevents a particular metadata deletion while known use remains, but physical data retention and actual Engine removal are separate operations. A delete method returning nil is not proof the local filesystem bytes were securely removed.

Topology Constraints During Placement

The standalone scheduler accounts volume topology and active writers from supplied tasks. Its focused workload test presents one running writable RWO task and one pending task sharing a volume, then expects no assignment for the second. This is a deterministic Plan constraint, not a live data migration experiment. Claim ch32.scheduler.

Ordinary service placement also has a real volume path, despite bypassing some resource/label planner filters. reconcileServiceTasksLocked resolves mounts, filters ready nodes through nodesForVolumeMountsLocked, and repeats the writer eligibility check as it creates assigned tasks. It calls attachTaskVolumesLocked immediately after storing each task. Therefore the volume constraint is not merely a disconnected planner test. Claim ch32.live-path.

The current MemoryService fixture creates a local volume explicitly bound to one ready node, a RWO claim and a service requesting two replicas. It observes only one created task and verifies its resolved VolumeID. It then injects task removal and checks attachment records for that task are detached. The actual focused suite passed both control-plane and scheduler volume cases.

That result is narrower than saying two-replica stateful services are supported safely. The desired replica count can remain two while only one writer is eligible. A replica deficit is an expected consequence of the storage constraint, not permission to ignore the writer rule. A UI should expose the reason rather than pretending a second ready replica exists.

State	Placement implication	Data implication
Unbound local volume metadata	First assignment can choose a node	No replicated bytes created
Bound mount with another node ID	Wrong node excluded	Data remains local
Active attachment and writable RWO request	Writer branch rejects another placement	No physical stale-host fencing
Task removed and attachment detached	Later logical use may become eligible	Volume bytes may remain

Binding snapshots need care. Resolved mounts carry the node value copied at resolution time. attachTaskVolumesLocked can subsequently set the volume record's NodeID. A task retry copying an old resolved mount or a multi-task loop retaining an earlier snapshot requires inspection to ensure the latest binding is represented. The current helper checks mount.NodeID rather than reloading the volume record for every predicate. This is a source-derived stale-snapshot edge, not a guarantee that every rescheduling history is already covered.

The Atlas failure window is node loss while a writer may remain alive. Starting a replacement on another worker would not bring over the local bytes and could create a second independent dataset. A correct operational response needs knowledge of the old process and actual data location. An offline heartbeat is not sufficient evidence for force-detach, migration or safe failover.

The Storage Guarantees That Are Absent

The repository's stateful-workload design proposes stable ordinals, ordered startup/shutdown, claim templates, snapshots, backup hooks and controlled replacement. Those are architectural ideas, not features to infer from the simpler VolumeClaim type. The current MVP does not implement a replicated stateful-service abstraction, distributed storage driver, automatic data migration or application-aware backup. Claim ch32.design-boundary.

A local cache and a database have different recovery needs. Losing a cache may be acceptable when it can be rebuilt; losing database bytes may not. The same RWO metadata predicate does not establish a backup strategy for either. Define the data's value, retention, rebuild process and acceptable downtime before choosing a replacement procedure.

Cleanup must state its retention choice. The deterministic chapter fixture creates only memory metadata, so process exit removes the test state and leaves no host volume. A live extension should default to retaining the exact named volume after removing its run-owned task, verify the data through an explicitly authorized subsequent mount if desired, and delete the exact owned volume only when disposable data removal is intended. Never use a broad volume prune to complete a chapter exercise.

The Engine adapter's container removal requests volume removal for associated anonymous volumes, while named-volume behavior has its own Engine semantics. Do not infer that every claim's named volume disappears with the container. Record the actual container mounts and volume inventory in a live retention test rather than relying on an unrelated metadata method.

The default server's attachment and volume maps are memory-owned. Restarting it can lose the logical inventory while physical named volumes remain on the daemon. This is another reason that memory-state recovery and storage-data recovery must be designed together. A fresh process can neither prove a volume is unused nor reconstruct all writer authority from an empty map.

History is the batched implementation 4d641bbe8913da568d6d9f4697a0144314e10b87. Its changed workload types, agent mounts, control-plane attachment logic and scheduler tests establish the MVP. The accompanying stateful design's broader promises remain design only. No separate implementation is invented for each original storage prompt.

For the safe Atlas lab, use the disposable source module and pinned Go 1.25 toolchain. Predict one writer for desired two, a refused conflicting scheduler assignment and detached metadata after the injected removal. Run:

```
go test ./internal/controlplane ./internal/scheduler \
  -run 'TestLocalVolumeSchedulingConflictAndCleanup|Test.*Volume' -
count=1 -v
```

The linked record contains the real selected tests and pass results. No PostgreSQL, Docker daemon, filesystem data write or network listener is needed. Cleanup is process exit. A wrong-node extension can present two fake nodes and a mount explicitly bound to one, then assert the other is ineligible; keep that result separate from the actually recorded baseline cases.

For a live storage experiment, use a unique owned named volume and harmless sentinel text, record the selected node and actual mount source, then distinguish container replacement from relocation. An observed sentinel surviving replacement on one node supports that one retention scenario. It would still not prove replication, backup, distributed fencing or isolation between equal names in different namespaces.

Exercises

1. Trace claim name, metadata VolumeID and physical Docker VolumeName. Which one reaches the bind string, and what namespace assumption can fail?

2. Explain the exact writable-RWO predicate when an active read-only attachment already exists. How does this qualify the documentation?
3. Why does the passing one-writer fixture not prove safe automatic failover after node loss?
4. Design a retention test with explicit cleanup, then identify the stale resolved-node snapshot case and the additional assertion needed before claiming all later placement stays pinned.

Worked answers preserve identity, topology and physical-effect distinctions. Local volume metadata and writer checks are implemented; replication, safe stale-host revocation and complete stateful recovery remain separate engineering work.

Chapter 33 – Extension Interfaces and a Plugin Roadmap

Atlas needs a second secret provider and a notification adapter. The tempting response is to build a plugin manager before understanding the interfaces already available. That would introduce process supervision, packaging, authentication, and compatibility work into an otherwise small integration. The first decision is therefore about the boundary: which side effect must vary, who calls it, and what survives a failed call?

By the end of this chapter, you should be able to locate existing extension seams, distinguish dependency injection from external plugin execution, and write a contract that rejects an incompatible adapter before it receives work. You will also learn why a runtime plugin remains privileged even when it lives in another process. The practical result is an executable fake adapter exercise and a deliberately unimplemented RPC proposal.

Interfaces Already Present in the Code

The runtime interface is a useful starting point because its methods describe observable effects. Pulling an image, creating a container, starting it, and inspecting it are separate operations. This lets the agent

recover after uncertainty between steps. A vendor adapter must preserve the meanings of those operations; merely returning successful HTTP responses would not satisfy the contract.

```
type Runtime interface {
        PullImage(ctx context.Context, image string, auth
*RegistryAuth) error
        CreateContainer(ctx context.Context, spec ContainerSpec)
(ContainerID, error)
        StartContainer(ctx context.Context, id ContainerID) error
        StopContainer(ctx context.Context, id ContainerID, timeout
time.Duration) error
        RemoveContainer(ctx context.Context, id ContainerID, force
bool) error
        InspectContainer(ctx context.Context, id ContainerID)
(ContainerStatus, error)
        ListManagedContainers(ctx context.Context, labels
map[string]string) ([]ContainerStatus, error)
        StreamLogs(ctx context.Context, id ContainerID, opts
LogOptions) (<-chan LogLine, <-chan error)
}
```

Exact source: internal/docker/docker.go:47–56.

The context on every method expresses cancellation and deadlines. It does not make an external operation transactional. If a create request reaches a remote engine but its response disappears, cancellation cannot prove that the container was never created. Task identity and a discoverable managed-container association are needed to resolve that uncertainty. The fake runtime's task lookup illustrates the replay boundary: repeated creation for the same task returns the existing identifier instead of adding another container.

Secrets expose a different seam. The Envelope interface accepts encryption and decryption requests, while the server constructs a local envelope and injects it into its memory control plane. This is an implemented, runtime-integrated boundary. A replacement provider could implement the same Go interface, but the current constructor

does not discover arbitrary remote secret plugins. Key selection, authenticated ciphertext, cancellation, and failure handling still need equivalent semantics.

The notification sender owns delivery retries and signing. Its HTTP client can be supplied through an option, and the dispatcher reads sinks through a narrow store interface. The server actually installs that dispatcher. An external webhook receiver is therefore a present integration surface, while an RPC notification plugin supervisor is a separate proposal. These distinctions matter when Atlas estimates the work: writing a webhook endpoint and building a secure executable loader have very different operational costs.

Autoscaling contains another instructive pair: a control-plane interface and a metrics-provider interface. The provider returns service CPU utilization rather than exposing a monitoring vendor's client. That makes controller policy independently testable. However, a clean interface is not evidence that the default server starts that controller. Trace constructors and loops before labeling a seam runtime-integrated. A compilable package can remain an implemented library with no production caller.

Boundary	Existing implementation	External-process status
Container runtime	Go interface, engine adapter, fake adapter	RPC protocol is design only
Secret envelope	Go interface, local authenticated encryption	Remote provider requires new composition
Notifications	Dispatcher and signed HTTP delivery	Webhook delivery exists; plugin supervisor does not
Autoscaler metrics	Injectable provider interface	Default server composition remains incomplete

The table follows the source, not a count of files named “plugin.” Commit 4d641bbe8913da568d6d9f4697a0144314e10b87 added jobs, storage, DNS, and notifications together. Its notification implementation is useful historical evidence. It does not establish that every design described in the later plugin document became an executable subsystem.

A Versioned External Plugin Handshake

The plugin design document proposes isolated processes and versioned RPC. It specifies a handshake containing protocol version, plugin type, implementation version, capabilities, and supported orch range. These are design requirements, not fields decoded by an upstream supervisor. Atlas can use them to define a reviewable interface without claiming the transport already exists.

Protocol version and implementation version answer different questions. Protocol version determines whether the two sides understand request and response meanings. Implementation version identifies which adapter build is running. Supported orch range prevents a transport-compatible adapter from silently relying on a newer workload schema. Capabilities describe optional behavior, such as log streaming or secret-key rotation, which callers must explicitly require before use.

For Atlas, a runtime adapter must advertise cancellation and task idempotency. An adapter with a matching major protocol but missing either capability is rejected before any container request. Unknown additive capabilities may be ignored. A major mismatch fails closed. This avoids optimistic fallback that would turn an unsupported operation into a partially executed workload change. The teaching test implements only this small validation rule; it does not start processes or send RPC messages.

Capability names need precise definitions. “Cancellation supported” might mean that a client stops waiting, that a queued request is discarded, or that an operation is interrupted before an effect occurs. Those are different guarantees. Atlas's proposed contract should say that cancellation stops new work where possible, returns a recognizable error, and leaves completed effects discoverable by idempotency key. It must not promise to undo a container that has already started.

A request identifier is useful for diagnostics, while an idempotency key represents the logical effect. Retries can have different transport request identifiers but the same task-create idempotency key. The adapter must either return the previously created object or report a conflict when the

repeated key carries a different specification. An interface that deduplicates only byte-identical requests needs an explicit canonicalization rule; map ordering and omitted defaults must not accidentally change identity.

Responses also need bounded size and stable error classes. Atlas should distinguish unsupported capability, invalid input, deadline exceeded, authentication failure, temporary unavailability, and unknown completion. A generic “retry” flag is insufficient when the effect may have happened. The caller's recovery decision depends on whether it can inspect by task identity, not merely on whether a network connection closed.

Supervisor Lifecycle and Failure Containment

A proposed supervisor has a sequence: validate configuration, connect or start, handshake, become available, accept bounded calls, drain, and stop. Each transition needs an owner and a timeout. Availability should begin only after handshake acceptance. During drain, new requests are rejected while existing requests receive a bounded completion interval. Expiry then triggers cancellation and process termination according to a documented policy.

This lifecycle is a teaching design. There is no upstream plugin supervisor to launch in this chapter. An implementation would need to decide whether executable paths are trusted deployment configuration, whether artifacts are signed, how Unix socket ownership is checked, and how mutually authenticated TCP identities are bound to configured plugin instances. Accepting a path from an ordinary workload request would cross an entirely different trust boundary.

Process isolation contains crashes and dependency conflicts, but it does not remove authority. A runtime adapter with Docker access can change host workloads; a storage adapter can expose or destroy data. Both remain in the trusted computing base. Filesystem restrictions, network policy, resource limits, and secret scoping must match the actual operations. A process with unrestricted host access and a private RPC socket is still highly privileged.

Atlas's most dangerous failure window is a successful remote create followed by a lost response and adapter crash. The supervisor may restart the process, but restarting does not settle whether the create succeeded. A safe caller inspects using the stable task key before issuing another create. The recovery path must be tested with an adapter that forgets volatile state while the underlying engine retains the object. The fake exercise here tests replay within one fake runtime; it does not implement that stronger cross-process persistence guarantee.

Backpressure matters just as much as crashes. An unbounded queue can turn a slow secret provider into a control-plane memory failure. Atlas should bound concurrent calls and queued work per plugin instance, expose rejection counts, and avoid holding a global orchestration mutex while waiting on a remote provider. A timeout does not solve queue growth when every request is admitted first. Retries should consume the same bounded budget rather than bypassing it.

Logging belongs inside the contract. The current notification redactor removes selected secret markers and sensitive detail keys, but pattern matching is not a universal secrecy guarantee. A proposed adapter should avoid accepting unnecessary credentials, return structured errors without secret values, and redact before writing process logs. Tests should use synthetic values and assert their absence without printing actual production material on failure.

Contract Testing a Proposed Adapter

The Atlas exercise uses the upstream FakeRuntime through the upstream Runtime interface, plus the local envelope implementation. It creates one fake container twice with the same task identity, checks equal identifiers, rejects work with an already canceled context, starts and stops the object, removes it, and checks that no fake managed containers remain. Separate cases reject a new protocol major, a wrong plugin type, and a missing required capability.

The envelope case encrypts a synthetic value with one local key, confirms another key cannot decrypt it, and confirms an already canceled request is rejected. These assertions exercise concrete implementation behavior. They do not certify a cloud key-management service, a hardware security module, or a remote adapter's authorization policy. The distinction keeps a small contract test useful without allowing it to stand in for every provider's acceptance suite.

Run the teaching file from a disposable copy beneath the orch module, because Go's internal-package rules apply. Copy it into `.book-labs/ch33/`, then run:

```
go test ./.book-labs/ch33 -count=1 -v
```

The recorded execution in adapter-contract.json passed. The command record retains its working directory, Go environment, UTC times, and output paths. It used no Docker daemon, database, network listener, external executable, or real secret. Cleanup means removing owned fake objects and letting test-local state disappear; it does not involve sweeping host containers.

To extend the experiment, first add a contract case, then build an adapter against it. A timeout adapter should distinguish a canceled wait from unknown effect completion. A malformed-response adapter should fail decoding before handing data to reconciliation. A crash adapter should leave a queryable operation record. These additional cases are proposed work, not results from the current run. Rejecting four handshake fixtures and replaying one fake create is narrower than supervising a production process through failure.

The chapter's boundary is therefore explicit: narrow Go interfaces and selected concrete adapters exist; the external RPC system remains a design. Atlas can adopt the tested seams today and use the proposal to estimate further engineering. It should not advertise third-party plugin support until transport, identity, supervision, compatibility, recovery, and provider-specific tests are implemented and observed together.

Exercises

1. Locate one runtime-integrated seam and one interface whose controller is not started by the default server. Identify the constructor or missing composition that determines the classification.
2. Run the adapter contract and change the compatible handshake to omit task idempotency. Explain why rejection must happen before container creation.
3. Model a create that succeeds remotely but loses its response. Specify the identifier used for retry and the query needed after adapter restart.
4. Review a proposal granting a storage plugin unrestricted host access because it runs outside the server process. Identify the unsafe assumption and define three concrete acceptance checks.

Worked answers are in Chapter 33 solutions. The claim ledger records source classification and execution scope.

Chapter 34 – Leader Election and the Durable HA Boundary

Atlas wants to restart a control-plane host without interrupting its API. Two server processes behind a load balancer look like an obvious solution. At the locked source revision, however, each default server creates its own MemoryService. Requests routed to different processes would see different services, nodes, credentials, and tasks. An extra endpoint would add inconsistency before it added useful availability.

This chapter separates API replication, durable state, and controller leadership. You will trace the lease interfaces, follow PostgreSQL session ownership, and identify what cancellation does and does not stop. The Atlas deliverable is a durable-composition decision record backed by deterministic leadership tests and a separate observed database lock boundary. It is not a demonstration of a highly available default server.

API Availability and Controller Leadership

API availability means that a surviving instance can serve the same authoritative state after another instance fails. That requires a shared durable control plane with equivalent transactional semantics, authentication state, and operational readiness checks. Controller leadership answers a narrower question: which process may run a particular mutating loop at this moment? A leader lock does not automatically make API handlers durable or mutually consistent.

The HA design proposes active API replicas and one active leader per controller. Independent lock names let one instance lead scheduling while another leads rollout progression. This avoids forcing every kind of background work through one process. It also introduces several independently observable ownership transitions. A dashboard showing “leader” without the controller name would conceal which work is actually protected.

The implementation exposes a compact lease abstraction:

```
type LeaderElector interface {
        Acquire(ctx context.Context, name string) (Lease, error)
}

type Lease interface {
        Name() string
        Done() <-chan struct{}
        Release(ctx context.Context) error
}
```

Exact source: internal/leadership/leadership.go:15–23.

Acquire may fail because another owner holds the name. Done signals that the current lease is no longer usable. Release is an explicit cleanup operation. A lease does not supply a monotonically increasing fencing token, and its interface does not prove that every downstream side effect checks ownership. Those missing properties are important when an old worker has already sent a request before losing leadership.

The local elector stores leases in one mutex-protected map. Its exclusion is meaningful only to callers sharing that elector instance. Two separate LocalElector objects do not coordinate across processes or even within the same process. It is a deterministic implementation for tests, not a substitute for a distributed authority. Atlas should use it to exercise wrapper behavior, then separately verify the database implementation and final server composition.

Source history preserves the distinction. Commit b29a6fe91e90f74721e5175e43bf12aed6410f19 added the HA design; eb3a07c20c550a533afaf36f414ede1ba6c00f78 added the leadership package. Neither changes the current fact that main constructs MemoryService and directly starts controller loops. Library implementation and executable integration remain different milestones.

PostgreSQL Advisory Locks and Connections

The PostgreSQL elector acquires a pooled connection, computes a key from the controller name, and calls `pg_try_advisory_lock`. If another session holds the lock, it releases the connection and returns ErrLockHeld. If successful, the lease retains the connection while monitoring it. This dedicated session is essential because the advisory lock belongs to the database session rather than to a Go object or a row.

The key derivation hashes `orch:` plus the controller name. It does not include an explicit cluster identifier. Two independently operated clusters sharing one database and the same names can therefore contend. Conversely, two supposed replicas connected to different databases can each acquire a lock and both act as leader. A matching integer key is useful only when every participant consults the same authoritative database session space.

Normal release asks PostgreSQL to unlock, then closes the lease and returns the connection to its pool. Cancellation follows a different path. The monitor sees the canceled context, invokes close, closes Done, and calls the pool connection's Release method. Returning a connection to a pool does not necessarily terminate its underlying PostgreSQL session. This difference is the chapter's central failure window.

The actual full-schema recovery experiment observed that cancellation closed the lease notification while a competing pool still could not acquire the same advisory lock. Closing the original pool finally allowed acquisition. The observation is scoped to that captured disposable database and teaching harness. It demonstrates why a local Done signal cannot be treated as proof that the database session lock has been released.

The release fast path returns immediately when Done is already closed. A deferred Release after monitor cancellation therefore does not issue a compensating unlock. The source explains the observed retained lock without inventing a PostgreSQL failure. Atlas must fix connection cleanup

in a separate implementation before relying on prompt handover; this book does not modify the locked application to make that guarantee true.

Monitoring also checks connectivity with `SELECT 1` every five seconds. A successful query means the connection answered that query. It is not a fresh acquisition, a fencing check against every workload mutation, or protection against unsafe database dual-primary operation. Timeouts, database topology, session termination, and write authorization must be evaluated together. A reachable database can still be the wrong authority for the intended cluster.

Observation	Supported conclusion	Unsupported conclusion
Acquire succeeds	This session obtained the named advisory lock	All API state is durable
Done closes	The local lease ended	Every old side effect has stopped
Callback returns	That wrapper invocation finished	Its earlier external requests were undone
Successor acquires	The consulted database permits a new owner	Two database primaries cannot both accept writes

Lease Loss Must Cancel Controller Work

RunWithLeadership validates its inputs, acquires once, marks the controller leader metric true, and defers cleanup. It creates a child context and cancels it when either the parent context or lease Done ends. Finally, it invokes the supplied callback with that child context. The wrapper implementation contains no acquisition retry loop. Retry with backoff belongs to a higher-level supervisor that must be designed and tested separately.

Cancellation is cooperative. A callback waiting on ctx.Done exits promptly in the upstream unit test. A callback ignoring its context can continue issuing work after the wrapper has signaled loss. Likewise, a database or runtime request already accepted before cancellation can complete

afterward. Atlas should pass the context through every layer, bound in-flight operations, and retain idempotent reconciliation so a successor can safely recompute uncertain work.

The deterministic loss scenario starts a wrapper, waits for its callback's started signal, calls LocalElector.Lose, and waits for context.Canceled. It then checks that the leader metric is false. This ordering is stronger than sleeping and assuming the loop has started. The channels establish causality: ownership was acquired, work began, loss was injected, and cooperative work stopped. The test does not simulate packet loss or observe a real database session.

A network partition introduces two timelines. The old process may lose database connectivity while still reaching agents. The database may eventually drop its session while an external request is still in flight. A new process can then acquire leadership. The safety question is whether repeated or overlapping work is harmless, not just whether two lease objects report different states. Atomic task claims, transactionally recorded events, and stable task identities remain necessary.

The proposed Atlas durable-composition ADR therefore requires cluster-qualified names, one verified database primary, explicit session cleanup, callback termination checks, and bounded reacquisition. It also requires database-backed API transactions before API replicas become ready. These requirements are future composition work; the ADR does not imply that upstream already supplies fencing tokens or a complete HA server.

The Durable State and Server Composition Prerequisites

Begin the lab in a disposable source copy with the locked Go toolchain. No live service is replicated. Run the upstream leadership package tests:

```
go test ./internal/leadership -count=1 -v
```

The recorded run passed the one-owner test, cooperative lease-loss test, acquisition-failure metric test, and stable-key test. Its artifacts preserve exact command, working directory, UTC interval, and output. Cleanup releases test-local leases; there is no Docker workload or database to sweep in this deterministic run.

Use the separate full-schema record for the database session experiment. Read its sequence rather than relabeling it as this unit-test run. It used owned disposable resources and tested competing PostgreSQL elector pools, cancellation, and eventual acquisition after the first pool closed. Its application store and backup checks are separate observations. The default server was not wired to that store merely because both packages appeared in the same harness.

The first acceptance gate for durable composition is equivalence of API semantics. Service creation must preserve its service record, initial version, and event together. Task assignment must retain its conditional pending guard and event atomicity. Credentials and namespaces must survive fresh-process restart under the same policy. A database pool injected beside MemoryService satisfies none of these requirements by itself.

The second gate concerns readiness. A durable replica unable to reach its authoritative database should not advertise that it can safely serve writes. A controller unable to establish leadership should not start mutating work. These can be different readiness dimensions: an API replica may serve reads while another instance leads a controller, provided the durable state and failure policy support that behavior. Atlas must define the policy rather than infer it from `/readyz` returning success today.

Finally, rehearse handover against realistic failure windows: orderly release, canceled controller, abruptly killed process, dropped database connection, wrong database identity, and uncertain external completion. Each case needs an observable owner transition and an invariant about

task/event effects. A green local lease test establishes a useful mechanism. It is only one piece of the evidence required for replicated operation.

A useful review artifact separates mechanisms from deployment assumptions. Record which database endpoint each candidate process uses, which controller name it requests, the UTC interval during which it believes itself leader, and the last accepted write associated with that interval. Redact credentials while retaining stable resource identity. This makes an overlap investigation possible without equating local metrics with database truth. If the old process stops logging, absence of a log line is not proof that its last runtime call failed. Inspect the task and the engine before allowing a successor to repeat the effect. That inspection is also how Atlas avoids turning a safe leadership handover into duplicated workloads.

Atlas leaves this chapter with a justified “not yet” for default-server HA and a concrete implementation plan. That decision protects the team from deploying two inconsistent memory control planes under an availability label. The implemented leadership package is valuable precisely because its contract and limitations can be inspected, tested, and improved without confusing it with the whole system.

Exercises

1. Explain why two LocalElector instances can both acquire `scheduler`, and identify the state they fail to share.
2. Run the leadership tests and trace the channel sequence that makes lease-loss cancellation deterministic.
3. Explain the observed retained advisory lock after cancellation. Distinguish pool connection release from database session termination and identify the fast path that prevents a later unlock.
4. Review a proposal to add two default servers behind a load balancer. Specify the durable-state, topology, readiness, and side-effect gates required before approval.

Worked answers are in Chapter 34 solutions. Source and observation classifications are preserved in the claim ledger.

Chapter 35 – Upgrades, Migrations, Maintenance, and Retention

Atlas schedules an upgrade for a quiet evening. The change plan names a new server binary, a database migration, and a retention cleanup. Treating these as one action would hide three separate risks: incompatible agents, destructive schema changes, and loss of diagnostic history. A successful installation command would answer none of those questions by itself.

This chapter teaches you to read pinned compatibility ranges, inspect migration locking and rollback guards, and distinguish a maintenance policy from an automatic shutdown of all work. You will also trace retention's preservation rules and the server's opt-in schema check. Atlas's lab combines actual disposable-database observations with deterministic operation tests and a controlled execution of the upgrade-preflight command.

Version Ranges and Upgrade Preflight

Compatibility is part of the release contract. At this source revision, the server is 0.3.0, the API is v1, supported agents range from 0.2.0 through 0.3.0, and the expected database schema is 16. The accepted schema range is 15–16. These are pinned constants, not promises about an arbitrary future release.

```
const (
        Server             = "0.3.0"
        MinimumAgent       = "0.2.0"
        MaximumTestedAgent = "0.3.0"
        Schema             = 16
        MinimumSchema      = 15
        MaximumSchema      = 16
)
```

Exact source: internal/version/version.go:11–18.

Agent versions below the minimum are classified too_old; versions above the maximum tested value are classified untested_newer. Empty agent version falls back to the server version in the checker. That fallback preserves older callers but makes missing version metadata weaker evidence than an explicitly observed binary version. Atlas should inventory installed binaries as well as the server's node records before a coordinated upgrade.

The parser expects three numeric version components and strips a leading v. Its limited handling of suffixes should not be mistaken for a complete implementation of every semantic-version precedence rule. The agent checker compares its parsed tuple against pinned bounds. A newer agent is a warning case, not a statement that its behavior has been tested. Compatibility metadata and executed interoperability tests answer different questions.

The CLI's check-upgrade implementation fetches version information and node records, classifies each agent, and emits a structured decision. It does not actually apply a migration, replace an agent, or inspect every database column. More subtly, a structured status of blocked is still returned through a successful output-writing path. A shell script that checks only the process exit code can therefore miss an upgrade rejection.

The controlled preflight fixture executes the real Cobra command against explicit node inputs. Agent 0.2.0 produced compatible with no warning; 0.1.0 produced blocked with a warning; 0.4.0 produced compatible with an untested warning. All command invocations returned no command error. This was observed in controlled-preflight-corrected.json, using an injected fixture rather than a live cluster. The initial fixture's nil embedded client failed and remains recorded; the corrected fixture initializes the client but performs no network calls.

Atlas automation must parse the decision and choose its own warning policy. For example, reject blocked, require review for untested versions, and proceed only after schema status and restore evidence pass

separately. Do not change the upstream output contract in the teaching harness. Learning to consume the actual interface is more useful than inventing a convenient exit-code guarantee.

Migration Locks and Guarded Down Paths

Migration discovery reads ordered filenames and groups up/down paths by version. Up acquires a database connection and executes under a PostgreSQL advisory lock. It checks recorded versions, skips those already applied, and applies each pending migration in its own transaction with its version record. This limits partial application within one migration; it does not turn an entire sixteen-migration upgrade into one transaction.

The transaction sequence records the version only with successful SQL application. If migration ten succeeds and eleven fails, earlier committed migrations remain. Atlas must inspect status after failure and decide how to repair forward or restore. Repeated up is useful because recorded versions are skipped, but idempotency depends on the version table and unchanged migration files. Editing an already committed migration changes the meaning of that history.

The lock wrapper attempts unlock even when the callback fails. A lock-acquisition failure prevents the callback from running. The runner's lock is separate from controller leadership locks discussed in the previous chapter: it serializes schema-changing migration runners, not every application write. Production planning still needs a policy for old and new binaries accessing the database while a migration runs.

Down rejects the request unless allow-down is true, then locates the newest applied migration and reads its down file. The explicit guard is an authorization boundary, not proof that the SQL preserves useful data. A file's existence does not make a rollback safe for customer records. Atlas should read both directions, identify dropped columns and tables, and prefer binary rollback against an additive compatible schema where possible.

The captured full-schema experiment began with no applied versions, observed sixteen pending migrations, applied them to current 000016, and observed no pending work on a second up. It also rejected down without authorization. No destructive down migration was necessary. The record includes separate backup/restore verification against owned databases, and it must not be relabeled as default-server restart recovery.

An earlier upstream store fixture applied only migrations one through fourteen. It then failed with missing agent_version, which migration sixteen introduces. Keep that original failure visible. A successful full-schema extension demonstrates that the later schema works for the tested package operations; it does not retroactively make the original fixture pass or prove that every accepted schema version supplies every currently queried column.

Maintenance Windows and Bounded Retention

Maintenance policy is selective. A window names allowed operations, a schedule, timezone, duration, and scope. The control-plane check first searches for relevant enabled windows. If none apply, the operation is allowed. If a relevant window currently permits the operation, it is allowed. Otherwise the operation conflicts unless a forced context requests an audited bypass. Creating an unrelated window does not freeze the cluster.

The time evaluator converts the current time to the configured location and searches scheduled occurrences within the preceding duration. Tests supply fixed time so that inside/outside decisions are reproducible. Atlas should examine daylight-saving transitions and its chosen timezone when creating operational schedules; a cron string alone does not explain which civil time operators mean. The implementation supports a bounded occurrence search, not a general calendar approval workflow.

Emergency force is also narrower than a universal permission override. It bypasses the maintenance-window check and appends a maintenance.bypass audit record. It does not automatically waive

workload admission, quotas, or runtime safety requirements. The operation test verifies an outside-window rollout rejection and an audited forced request. The recorded run passed, but no real host was drained or upgraded in that deterministic fixture.

Retention must preserve the state needed to operate and diagnose the system. Defaults keep events for thirty days, audit logs for ninety, completed tasks for seven, failed tasks for thirty, rollouts for thirty, and completed jobs for seven. These values are configuration defaults, not legal retention advice. Atlas should choose them according to its recovery and investigation needs, then verify the actual pruning branches.

The task pruning logic considers stopped/removed tasks and resolved failures. An old running task is preserved. An old failed task is preserved when no qualifying replacement or cleanup intent resolves it. The resolution helper accepts a running or healthy task for the same service at an equal or newer version, or stopped/removed desired state. Those are code-level rules; “resolved” is not a human incident status.

Rollout pruning removes old succeeded or rolled-back deployments, and job pruning removes old succeeded jobs. Failed unresolved histories stay available under these branches. GitOpsRecords appears in retention configuration, but this pruning method has no corresponding GitOps-record deletion loop. Configured fields must not be reported as enforcement until the caller and mutation path exist. Similarly, pruning control-plane task records does not remove arbitrary engine containers.

Dry-run computes eligible counts while retaining the collections. The retention fixture checks an old event, active task, unresolved failure, and resolved failure. Actual execution passed dry-run preservation and subsequent selected deletion. This is a bounded semantic test, not a performance measurement against years of history. The implementation holds the memory-service mutex while scanning; large retained collections can affect concurrent request latency.

A Schema Check Must Be Enabled to Run

A database URL in configuration does not itself activate schema validation. The server checks ORCH_SCHEMA_CHECK only when its value equals true ignoring case. It opens the migration runner, reads current status, checks the supported range, and then proceeds to construct MemoryService. A successful check says something about the database schema it inspected. It does not switch active state into PostgreSQL.

For a source-checked future procedure, enable the gate explicitly:

```
ORCH_SCHEMA_CHECK=true orch-server
```

The command assumes the configured database, migration directory, credentials, and other required server settings are already supplied. It is a procedure, not a claimed execution from this chapter. Values such as 1 do not satisfy this particular comparison. Leaving the variable unset skips the check, which is important when reading a development startup log that looks healthy despite an unavailable database.

Atlas's safe lab has three independent evidence lanes. The operations run exercised pinned version checks, migration lock/discovery helpers, fixed-time windows, and memory retention preservation. The controlled preflight exercised real CLI decision output with fixture nodes. The already captured disposable-database experiment exercised actual status, up, repeated up, guarded rejection, and restore. None should borrow another lane's coverage.

The cleanup plan should be equally explicit. In the deterministic lane, only test-local data exists. In the database lane, use the captured database names and container identity, verify that they were created for the experiment, and remove only those resources. A command such as dropping every database matching a loose prefix would exceed the experiment's ownership. Keep the restore target separate from the source so a successful verification cannot be confused with reading

unchanged original rows. Preserve evidence before cleanup and record cleanup failures as part of the result rather than hiding them behind a passing migration status.

Before a change window, preserve the candidate binary identity, compatibility metadata, actual node versions, current/pending schema, tested backup reference, key availability, maintenance decision, and retention preview. Afterward, capture resulting status and compare intended effects with actual records. If a server restart loses memory state, restoring a database dump cannot reconstruct records that were never written there. That boundary remains true even when migrations, backup scripts, and compatibility checks all succeed.

Exercises

1. List the locked server, agent, API, and schema bounds. Explain why expected schema sixteen and accepted schema fifteen are different statements.
2. Run the controlled preflight fixture. Design automation that rejects structured blocked status even when the command itself succeeds.
3. Trace a migration failure after earlier migrations committed. Explain what status and backup evidence you need before retrying, without executing down on real data.
4. Classify old running, unresolved failed, and replaced failed tasks for retention. Identify why an unrelated maintenance window and a GitOps retention setting do not establish the intended enforcement.

Worked answers are in Chapter 35 solutions; the claim ledger preserves the distinction between inspected procedures and executed observations.

Chapter 36 – Fakes, Property Tests, Fuzzing, and Contracts

Atlas's test suite is green, yet the team has already observed a resource request disappear between its API and Docker. The lesson is not that tests are useless. It is that a passing test answers the question encoded by its fixture and assertions. A fake runtime can validate reconciliation order while overlooking a field that nobody asserted. A parser fuzz target can survive malformed input while accepting a value with the wrong meaning.

This chapter teaches you to read tests as executable contracts. You will separate fake behavior from runtime behavior, reproduce seeded scheduler properties, run a bounded fuzz campaign against an actual target, and choose an independent assertion for an orchestration boundary. The Atlas result is an evidence-backed testing plan that records what passed, what was skipped, and which important claims remain outside each test's scope.

Fakes as Executable Specifications

The runtime fake implements the same narrow interface used by the agent. It keeps container state, operation history, created specifications, and injectable errors under a mutex. This makes it useful for assertions about sequencing, repeated work, and concurrent access. The creation implementation deduplicates by task identity, while separate methods model start, stop, inspect, and removal. Those choices define the simulation the tests inhabit.

A fake is not automatically a faithful engine. It does not allocate Linux namespaces, enforce cgroups, pull image layers, bind real host ports, or reproduce every Docker daemon error. Tests using it should say which fake transitions are relied upon. If Atlas needs proof of a memory limit,

inspect the ContainerSpec supplied to the fake and then separately observe a real engine's configured limit. A successful fake start is insufficient.

The upstream fake-runtime tests check recorded operations and concurrent same-task creation. The repeated identifier assertion is an idempotency contract. The concurrent case tests mutex-protected behavior within one fake instance. Neither proves deduplication across independent engine adapters after process failure. That stronger claim needs stable managed labels, engine inspection, and an actual uncertainty/retry scenario.

The end-to-end test uses a real handler, client, agent Runner, and fake runtime. Its default entry point selects the fake explicitly, while real Docker is separately gated by ORCH_E2E_DOCKER. It exercises deploy, logs, scale, and deletion through multiple components. Calling it an integration test is justified by those relationships; calling it a live container deployment would be false when the selected runtime is fake.

Contracts also include the fixture's assumptions. The end-to-end log assertion looks for a fake-container start line. A real nginx image does not necessarily emit that text. Setting a Docker flag without reading the shared assertion can therefore fail for a harness reason rather than a workload failure. Preserve the failure and distinguish an adapted real-runtime harness from the original test. Do not silently weaken assertions and retain the old coverage label.

Commit c69f9ffa830284692aabf9e65ab9eb1f79d3a114 introduced the fake boundary; 903ece1ea30c5a0f4902b3f2e9cb475fdea41fdc added the MVP flow. Read their changed code alongside current assertions. A historical test name is evidence of intent and implementation history, not proof that it ran in the present environment.

Seeded Scheduler Properties

Example-based tests pick a specific placement and compare its result. Property tests generate many inputs and assert invariants that must hold across them. The scheduler's randomized invariant test uses five fixed seeds: 1, 7, 42, 99, and 20260821. Each subtest constructs nodes with varied readiness, labels, capacity, service requests, and occasional published ports. Fixed seeds make a failed state reproducible.

The same input is planned twice and compared for determinism. Assignments are then checked for unique task IDs, ready nodes, matching placement labels, aggregate CPU/memory bounds, and per-node port uniqueness. This is stronger than expecting a particular distribution from one hand-picked cluster. It still covers a finite generated domain, and it exercises Plan rather than every active assignment path in the server.

```
                        first := Plan(input)
                        second := Plan(input)
                        if !reflect.DeepEqual(first, second) {
                                t.Fatalf("seed %d:
nondeterministic", seed)
                        }
                        nodeByID := map[types.NodeID]types.Node{}
```

Exact source: internal/scheduler/property_test.go:46–51.

The failure messages retain the seed, allowing Atlas to rerun the precise subtest. Keep the seed and the source revision together: changing input-generation code can produce a different cluster from the same integer. For a durable regression, preserve the minimized concrete input as well as the seed. A random number is a reproducibility aid, not a permanent specification independent of the generator.

Coverage also depends on input shape. The property fixture starts with generated pending tasks and no occupied-node task inventory. It does not thereby cover every preexisting allocation, reservation, or concurrent store race. Atomic assignment remains a different layer. A planner can

propose a legal assignment and still lose its claim when another scheduler commits first. That conflict should trigger recomputation, not be treated as a planner-property violation.

Atlas's independent review question is whether the invariant matches the operational claim. Capacity safety should sum requested resources by selected node, not compare the output against another copy of the planner's scoring function. Port safety should inspect collisions in the resulting assignments and occupied ports, not merely assert that a helper returned true. A test that repeats implementation logic can preserve the same mistake on both sides.

The seeded execution record passed all five generated subtests, concurrent fake creation, and selected repeated-reconciliation tests. These were actual uncached executions. They used the locked disposable source and no Docker or database. Package and subtest pass lines are retained separately so their count is not inflated into a claim about unique production scenarios.

Fuzz Inputs and Minimal Reproductions

Fuzzing explores malformed and unexpected byte sequences using coverage feedback. The actual port target is named FuzzPortSpec. The testing document instead gives an example using FuzzParsePortSpec. Atlas must resolve that name against source before copying the command. This small documentation conflict is recorded rather than repaired inside immutable evidence.

The port target seeds ordinary mappings, UDP, zero, an out-of-range number, and arbitrary text. Its callback ignores both parsed values and returned errors. The resulting property is primarily that the parser does not panic on supplied inputs. It does not assert that every accepted port is semantically safe or that rejected ports receive an ideal error message. Those require explicit table or property assertions over the result.

The same distinction applies to service YAML and resource quantities. A parser may return an error for malformed input and correctly satisfy a panic-resistance target. Conversely, it may accept a surprising value without violating that target. Atlas should pair fuzzing with meaningful semantic checks such as bounds, round trips under a defined canonical form, or rejection of a known unsafe syntax. Do not describe ignored return values as comprehensive validation coverage.

The bounded experiment used one worker and a two-second campaign:

```
go test ./pkg/types -run '^$' -fuzz '^FuzzPortSpec$' -fuzztime 2s -
parallel 1
```

The actual fuzz record passed after 42,919 executions, with five baseline seeds and sixty-three newly interesting inputs reported. These are observed figures for this run. “Interesting” means useful coverage feedback, not a confirmed bug or a distinct real-world port configuration. No crash was discovered, so there is no invented minimized failing input to present.

The command deliberately excludes ordinary tests, selects one exact fuzz target, bounds duration, and limits parallelism. This keeps the exercise predictable on the shared machine. A two-second campaign is a smoke observation, not an argument that all possible inputs are safe. The command record's toolchain, source pin, and UTC interval make it possible to repeat the experiment under a larger approved budget later.

If a future campaign fails, preserve its original record before editing anything. Reproduce the generated input with the exact target, inspect whether the failure comes from the parser or the harness, and minimize it without losing the invariant. Then add a stable regression case in a separate teaching extension or an authorized application change. The book's immutable snapshot must remain unchanged, even when the discovered behavior is inconvenient.

Contracts Across Boundaries

Atlas's resource-loss incident suggests a test that belongs between two existing layers. The API accepts requested CPU/memory, the service stores it, the agent receives a task DTO, and Docker receives a ContainerSpec. Checking only the service record cannot prove that the runtime received the fields. Checking only the engine adapter with a manually complete ContainerSpec cannot prove that the normal agent path supplied them.

The observed real-Docker boundary experiment did both and found different outcomes: direct adapter limits were present while the ordinary handler/Runner path yielded zero runtime limits. This is a contract failure in propagation, not evidence that Docker cannot enforce the limits. A good regression would assert the same requested values at each handoff and then inspect the actual engine configuration. Until such an upstream fix exists, the book preserves the limitation.

Repeated convergence deserves similar care. The reconciler test reruns reconciliation and checks duplicate work, but process restart requires identifying which state survives. Recreating a controller around the same store tests recomputation against retained state. Creating a fresh MemoryService tests a different boundary. Atlas should name both the restarted component and its retained dependencies in every result.

Test form	Useful evidence	Boundary to retain
Fake adapter contract	Call order, supplied fields, replay behavior	No kernel/container enforcement
Seeded planner property	Finite generated invariants and determinism	Not concurrent persistence
Parser fuzz target	Bounded exploration of callback assertions	Ignored results are not semantic proof
Cross-layer acceptance	Values and effects across real handoffs	Scope depends on selected adapters

The full baseline suite and race detector are separate captured evidence. Their explicit Docker/PostgreSQL skips remain skips. A passing ungated suite does not certify the gated paths, and a clean race run detects only races reached under its execution. Combining those records with targeted contracts gives a more honest confidence statement than counting green checks.

For the lab, begin with a disposable locked copy and clear live integration variables. Run `go test ./internal/scheduler -run '^TestSchedulerRandomizedInvariants$' -count=1 -v` for all five seeded cases, then run the bounded fuzz target. Observe subtest names and rejected/accepted outcomes rather than expecting an invented terminal transcript. No host containers or databases are created. Fuzz corpus caches belong to the configured disposable cache; retain useful failures explicitly before deleting owned temporary files.

Atlas should now ask a sharper question in review: which invariant would fail if this boundary dropped, duplicated, delayed, or renamed a value? That question turns fakes, properties, and fuzzing into complementary tools. It also prevents a passing simulation from becoming an unsupported claim about durable recovery, kernel isolation, or production capacity.

Exercises

1. Identify one fake behavior that supports a useful agent test and one real Docker behavior it cannot establish.
2. Run the seed_42 scheduler subtest and explain why the seed must be recorded with the generator revision. Add a proposed occupied-port input without duplicating the scoring algorithm as the oracle.
3. Run the bounded port fuzz command. Explain exactly which failure its callback can detect and why newly interesting inputs are not bug counts.

4. Design a resource-propagation acceptance test across API, agent DTO, ContainerSpec, and engine inspection. Explain why testing only the direct adapter would miss the observed gap.

Worked answers are in Chapter 36 solutions, with scoped claims in the evidence ledger.

Chapter 37 – Failure Injection and Recovery Evidence

Atlas has a service with one healthy replica and an upgrade waiting to roll out. Before approving the change, an operator asks what happens if the agent disappears after starting the replacement. “The chaos tests pass” is not yet an answer. The test might replay a status report, reconstruct a controller around retained memory, or actually terminate a process. Each experiment crosses a different failure boundary.

This chapter teaches you to choose an invariant before injecting a fault, identify the state retained by a simulation, and build a recovery timeline from observed events. You will inspect the upstream chaos cases, run selected deterministic failures, and compare them with the actual fresh-server/Docker experiment. The Atlas outcome is a small failure dossier whose conclusions remain proportional to the fault that was injected.

Define the Invariant Before the Fault

A useful drill begins with a specific statement that should remain true. For an agent retry, the same task should not acquire a second managed container. For a failed scheduler write, no assignment event should claim an assignment that never persisted. For service deletion during rollout, desired cleanup must take precedence over new workload progression. These are separate invariants with different observation points.

The upstream chaos suite starts with repeated running reports against one memory control plane. That case is named “agent restarts during task start.” Its actual injected condition is a repeated status report using the same container ID. It checks that one running task remains. No agent process is killed, no engine object is inspected, and no volatile agent state is reconstructed. The test is useful for report idempotency within its stated scope.

The case named “server restarts during rollout” likewise retains the same control-plane object. It constructs a rollout controller and runs one pass against existing service and deployment records. This tests recomputation from retained state, not persistence across a fresh server process. The relevant code contains no new MemoryService after the rollout begins. Atlas must not turn the subtest's name into a stronger durability claim.

This precision is an operational advantage. If a later incident loses service records after a process restart, the team can immediately see that the controller-reconstruction test never covered that boundary. A misleading resilience label would instead send investigators looking for a regression in a guarantee that did not exist. Fault names should describe both the failed component and the retained state.

Commit fe385251b7a2051b10af3dd6caaad8bb4821fa21 introduced these deterministic cases. Earlier 66cfb7049fc1fbceeec9e36a7067febe8cbab698 hardened agent recovery and failure reporting. Their changed hunks explain how recovery logic evolved; the present chapter's executed records establish which selected tests ran now.

Runtime Failure, Node Loss, and Partitions

A failed inspection is not equivalent to a missing container. The agent's unavailable-runtime test injects errors into both inspect and managed-container listing, then asserts that no new status reports were made and

no create call occurred. The desired invariant is conservative behavior while reality cannot be observed. Creating a replacement immediately could duplicate a workload whose original container is still running.

```
        if err :=
runner.reconcileAssignedTasks(context.Background(), nodeID); err !=
errDockerUnavailable {
                t.Fatalf("expected docker unavailable error, got
%v", err)
        }
        wantRuntimeCalls := []string{"inspect:container-1", "list",
"list"}
        if !equalStrings(runtime.OperationStrings(),
wantRuntimeCalls) {
                t.Fatalf("expected runtime calls %#v, got %#v",
wantRuntimeCalls, runtime.OperationStrings())
        }
        if len(client.statuses) != 0 {
                t.Fatalf("expected no status reports while Docker
availability is unknown, got %#v", client.statuses)
        }
```

Exact source: internal/agent/agent_test.go:198–207.

The fake options permit pull, create, start, inspect, and list failures to be chosen independently. This makes the injection point explicit. A pull failure occurs before a new container exists; a start failure may leave a created container; an inspection failure leaves existence uncertain. Atlas should select the option that represents the intended failure window rather than use a generic error everywhere.

The broad chaos suite's “docker runtime fails to start container” case reports a failed status with a reason directly to the control plane. It checks storage and event behavior after that report. It does not execute the Docker adapter's StartContainer method. Selected lower-level agent/runtime tests provide the missing call-path evidence. Combining layers is useful only when the report identifies which layer supplied each observation.

Node loss introduces a harder distinction between authority and reality. The server can mark a node offline and remove a stateless task from its active assignment set. The isolated host may still run its old container. Replacement on another ready node can temporarily coexist with that old workload. The return path can eventually remove the stale managed container, but eventual cleanup does not establish zero overlap during a partition.

The returned-node test supplies no active assignment and a managed stale container. It asserts list, stop, remove, and a removed report. This proves the fake-runtime cleanup behavior for the modeled task on return. It does not provide network fencing, volume fencing, or a proof that two replicas never served requests simultaneously. Stateful recovery requires stronger ownership rules and operator decisions beyond that fixture.

Ownership labels are part of safe cleanup. The agent's separate non-managed-container test keeps a foreign object intact. Atlas drills should likewise create a captured object with exact test identity and check cleanup against that identifier. Broad commands deleting all nginx containers or all objects with a partial name are inappropriate on a shared host. Cleanup is an assertion about ownership, not an afterthought after the interesting fault.

Recovery Timelines and Residual Work

A recovery timeline has at least four boundaries: the fault injection, detection, corrective intent, and observed completion. These may be in different components. A server heartbeat timeout changes its view of a node; the agent on that node may know nothing about that change until connectivity returns. A task marked removed in API state is not automatically an engine observation that its container has stopped.

The suite's scheduler-write case injects an AssignTask error and checks the event slice remains empty. That assertion prevents a false assignment history. It does not emulate a PostgreSQL transaction rollback, database

failover, or ambiguous commit response. The fake store returns a deliberate error before the successful-assignment path. Actual concurrent claim and transaction behavior need separate database tests.

Deletion during rollout checks that the service reaches deleting or deleted and records deletion-started history. This is an intent and control-plane state check. It does not assert every container is already gone at that point. Atlas should distinguish accepted deletion, task cleanup directives, agent removal, and terminal service state. The residual work between those stages is exactly what an interrupted cleanup drill should inspect.

For the executed deterministic lab, use a disposable locked source copy and run:

```
go test ./internal/e2e ./internal/agent ./internal/docker \
  -run 'Test(ChaosScenarios|
ReconcileAssignedTaskDoesNotRecreateWhenDockerUnavailable|
ReturnedNodeRemovesNodeLostUnassignedContainer|
FakeRuntimeFailureInjection)$' \
  -count=1 -v
```

From the disposable pinned module, run `go test ./internal/e2e ./internal/agent ./internal/docker -run 'Test(ChaosScenarios|ReconcileAssignedTaskDoesNotRecreateWhenDockerUnavailable|ReturnedNodeRemovesNodeLostUnassignedContainer|FakeRuntimeFailureInjection)$' -count=1 -v` with live integration variables cleared. The actual fault-test record passed all eight chaos subcases and the selected agent/fake assertions. It used no Docker daemon, database, real node partition, or killed application process. Its zero-looking printed test durations are rounded framework output, not measured recovery objectives. No production recovery-time claim follows from those lines.

The independent combined restart experiment crossed a real process boundary. It manually created a cached nginx container with no published ports, associated its exact ID with a task through an authenticated report, stopped the first server binary, and started a fresh

second server. The second process had empty service/task/node lists while the same captured container remained running. The owned container was then removed and absence verified.

That experiment is intentionally scoped. The agent Runner did not create or adopt the container; the harness supplied the association. Its purpose was to observe the relationship between lost memory state and a surviving engine process. It cannot be cited as automatic agent recovery or PostgreSQL restoration. It does establish that restarting the default server and waiting for healthy HTTP does not restore the earlier control-plane records.

Drill	Fault or modeled condition	Evidence boundary
Repeated report	Same running report twice	One memory task remains
Controller reconstruction	New rollout controller, retained store	Progress recomputed from existing records
Runtime unavailable	Fake inspect/list errors	No speculative replacement or status
Returned node	Stale managed fake object, no assignment	Stop/remove cleanup on reconciliation
Fresh server	Actual first process stopped, second started	Memory state lost while captured Docker object survives

The table is a failure map, not a ranking of test quality. Deterministic fixtures make narrow invariants cheap and repeatable. Process experiments expose ownership and persistence boundaries. Real runtime experiments expose kernel and daemon behavior. Atlas needs all relevant kinds without pretending that one can replace the others.

The Scope of Fake Chaos

A chaos experiment should preserve a useful result even when recovery fails. Record initial IDs, the exact injected condition, expected invariant, actual observations, error path, and cleanup. Do not delete the failed run

because a later harness revision passes. The original failure can reveal a misunderstanding of the test or a real implementation boundary. A corrected run should identify precisely what changed.

Atlas's prior real-Docker flow illustrates this discipline. A first log-check guard inspected the wrong task association and failed. The corrected harness scoped log retrieval to the captured task and passed deploy, scale, logs, and deletion. Both records were retained. Describing only "Docker passed" would hide the harness correction and prevent a reader from understanding which behavior was actually tested.

A safe escalation from fake to live fault should add one uncontrolled boundary at a time. Start with a deliberately failed runtime call and an exact operation-history assertion. Next use an owned disposable engine object with no published host ports. Then, if authorized and supported, test a process or network failure while retaining captured identities. A live shared production node is not a convenient substitute for missing disposable infrastructure.

Recovery acceptance also needs a stopping rule. If the desired state cannot be recovered because the default control plane is empty, repeatedly restarting the same binary will not create durable history. Stop the drill, preserve evidence, clean owned objects, and record the architectural prerequisite. For Atlas, that prerequisite is durable server composition, not a longer polling timeout.

A drill owner should also record who decides that recovery has completed. API task state, agent acknowledgement, and engine inspection are distinct witnesses; none should silently substitute for another. When they disagree, retain all three observations and investigate the interval rather than selecting the most reassuring result.

The final reliability statement should be concrete: selected fake failure paths preserve their asserted state/event rules; the actual fresh-server experiment loses memory state; returned-node cleanup is eventual and does not fence a partitioned workload. These findings guide engineering

priorities without inventing production guarantees. A small honest failure dossier is more useful than a broad resilience label whose test boundaries nobody can explain.

Exercises

1. Rename two upstream chaos cases so their titles state the actual modeled action and retained state. Explain which stronger claim the original names might invite.
2. Run the selected fault tests and identify the operation history proving no speculative container create during runtime unavailability.
3. Draw a written timeline for node partition, stateless replacement, and returned-node cleanup. Identify the interval in which an old container may remain live.
4. Compare controller reconstruction with the combined fresh-server experiment. Define the evidence required before claiming durable rollout recovery and a cleanup plan that touches only captured resources.

Worked answers are in Chapter 37 solutions. The claim ledger distinguishes source assertions, simulated faults, and actual process observations.

Chapter 38 – Metrics, Dashboards, and Actionable Signals

Atlas's dashboard shows a growing “Task State Counts” panel while the current service has only two replicas. Nothing is necessarily wrong with the workload. The panel sums a counter of reported state changes, so it accumulates transitions rather than displaying current task inventory. Before using the panel to trigger an incident, the team must understand what its metric actually measures.

This chapter follows a signal from emission to exposition, scrape configuration, alert expression, dashboard query, and operator action. You will distinguish counters from gauges, detect misleading proxy names, and identify when a collector exists without an active caller. Atlas's lab injects one synthetic runtime failure and observes the real metrics wrapper and registry output. Alert evaluation and live Grafana operation remain separate, unclaimed activities.

Metric Emission and Meaning

Metrics begin at a call site, not a dashboard. The server registry defines API request counters and duration histograms, scheduler/reconciler collectors, task and rollout counters, autoscaling gauges, leadership gauges, and DNS counters. The collector definitions provide exact names and labels. Their presence establishes an available instrument; it does not prove that a particular executable path emits observations into it.

The server actually supplies metrics to its scheduler and rollout controller. API middleware records recognized method/route/status combinations, and the agent binary wraps its runtime with instrumentation. In contrast, a registered reconciler or leadership

collector does not mean the default server starts those components through that instrumented path. Read the constructor and loop together before interpreting a zero value as successful idle operation.

The access-log middleware records only paths recognized by metricRoute. Route normalization is an important cardinality boundary: a route template should represent many object IDs. Using raw user-controlled paths as labels would create new time series for arbitrary URLs. A missing unrecognized path in this metric is therefore not proof that no HTTP request occurred; logs and request metrics have different selection rules.

Even an apparently simple counter needs a precise event definition. The task-failure metric increments after an accepted agent request whose requested status is failed. The call site does not test whether this is the first failed transition. Duplicate accepted failure reports may therefore increment the counter while the control-plane event logic remains idempotent. Treat tasks_failed_total as counted accepted failure reports, not an exact count of distinct failed task identities.

The Docker wrapper has a particularly clear definition:

```
func (r *instrumentedRuntime) measure(operation string, call func()
error) error {
        r.metrics.IncDockerOperation(operation)
        err := call()
        if err != nil {
                r.metrics.IncDockerOperationError(operation)
        }
        return err
}
```

Exact source: internal/docker/metrics.go:96–103.

It counts the attempted operation before calling the next runtime, and counts an error when that call returns a non-nil error. It does not measure a container's eventual application health. A successful start call can still be followed by an immediate application exit; healthcheck and task-status observations belong to separate signals.

Commit b3e29e593f46b4e90899a56bd9391c03d7c42f1a introduced observability code. Later alert and dashboard commits added consumers. History helps explain why collectors and panels evolved separately, but the current source and actual emission determine the meaning available to Atlas now.

Labels, Cardinality, Counters, and Gauges

A counter accumulates observations and normally resets when its process restarts. A gauge describes a sampled value that may increase or decrease. A histogram records observations across cumulative buckets with count and sum. Confusing these types changes the question a query answers. Summing a state-change counter cannot reconstruct the current number of running tasks after arbitrary transitions and restarts.

Agent metrics label Docker operations by operation and task state changes by status. These are bounded vocabularies. API labels use method, normalized route, and status. Leadership uses controller names. The autoscaler recommendation gauge uses service_id, which is a deliberate higher-cardinality exception to the general low-cardinality guidance in the observation documentation. Atlas should budget and review that series lifecycle rather than claim every collector avoids object identifiers.

Labels also define aggregation boundaries. Summing Docker errors across every agent can identify a broad runtime problem but hides which instance failed. A dashboard grouped by operation can distinguish image pulls from start failures; an operator then needs instance-level detail or logs to find the node. Do not add secret-bearing image URLs or arbitrary workload labels simply to make a convenient drill-down.

The registry handler exposes Prometheus text for its own registry. Separate server and agent registries produce different series. A successful server scrape does not establish that the agent's runtime metrics were

collected, nor that PostgreSQL exporter credentials work. Atlas should verify each configured target and retain the job/instance labels added by Prometheus.

The bundled scrape configuration uses fifteen-second scrape and evaluation intervals. It names server, agent, and postgres jobs with explicit container-network targets. Those names matter because availability rules filter by job. Copying only an expression into another Prometheus deployment with different job labels can silently produce no series. Configuration is part of the signal chain.

Alert Expressions and Dashboard Proxies

The Docker error alert is named DockerOperationErrorsHigh. Its exact expression is `sum(rate(docker_operation_errors_total[5m])) > 0.1`, held for ten minutes. The corresponding dashboard panel sums the same rate grouped by operation. The rule aggregates across agents, whereas the panel preserves the operation dimension. Atlas's first response should inspect affected agent logs, Docker daemon availability, socket permissions, and the failed operation type.

One failed pull in a unit test does not make that alert fire. A rate requires a time series across samples, and the rule's for interval requires the condition to remain true. The chapter's lab observes an increment and text exposition; it does not run Prometheus's evaluator for ten minutes. Keep that missing observation explicit rather than presenting a fabricated alert screenshot.

Other rules are proxies with potentially surprising names. NodeOffline checks `up{job="orch-agent"} == 0`, meaning Prometheus cannot scrape the agent target. It is not the control plane's authoritative node-offline condition. DatabaseUnavailable similarly checks exporter scrape availability rather than directly testing database transactions. A failed scrape may result from endpoint, credentials, routing, or process failure.

The heartbeat rule checks `increase(heartbeat_success_total{job="orch-agent"}[2m]) == 0` for five minutes. An absent series is not automatically the number zero. A broken scrape path can remove the heartbeat observations while a separate up rule detects target failure. Atlas needs both data availability and behavioral progress signals. Reading one expression literally prevents assuming that it covers every missing-heartbeat situation.

The dashboard's Task State Counts panel sums task_state_changes_total by status. Its Node Status Counts panel counts successful and failed agent scrapes. Its rollout “success” expression subtracts failures from accepted rollout/rollback requests over an hour. These are useful operational proxies if labeled accordingly; none is a direct current-state inventory or a verified count of completed successful deployments.

The API error-rate rule divides 5xx request rate by a denominator clamped to at least one request per second. At low traffic, that differs from the literal fraction of failed requests. A single failing low-volume endpoint can produce a lower displayed ratio than an operator expects from counting requests. Atlas should understand this choice and test any proposed replacement expression against intended low-traffic behavior before changing alert policy.

Artifact	What it observes	What operators must verify separately
Docker error counter	Runtime calls returning errors	Daemon state and affected workloads
Agent up series	Scrape reachability	Control-plane heartbeat/node status
State-change counter	Reports by status	Current task inventory
Rollout request minus failure query	Windowed request/failure counters	Terminal deployment outcomes

Commit 357903ef7cb95a7d6d8fad5f426a48ed48bc0b76 added alerts; aa44e586fd79147173588f0bf86763814bd23ec4 added the starter dashboard. The source-pinned artifacts remain starting points. Their existence does not establish an Atlas SLO, a live monitoring installation, or tested alert routing.

Operator Action and the Atlas Signal Lab

The teaching fixture constructs an actual agent registry and wraps an upstream FakeRuntime with the actual instrumentation wrapper. It injects a synthetic pull error, invokes PullImage once, and exposes the registry through an in-process HTTP recorder. No listener, Docker daemon, database, or external monitoring service is involved.

Run it from a disposable module overlay:

```
go test ./.book-labs/ch38 -count=1 -v
```

The recorded execution passed. It observed docker_operations_total with operation pull_image equal to one and docker_operation_errors_total with the same label equal to one. Those are actual collector outputs from an actual wrapped call, with the runtime failure deliberately simulated. The separately captured upstream metrics tests exercise collector exposition for additional instruments.

The complete conceptual chain is now inspectable: the wrapped pull returns an error; measure increments the operation/error counters; the agent registry exposes them; the scrape configuration selects the agent endpoint; the alert consumes an error rate; the dashboard groups by operation; the operator investigates image-pull failure causes. Only the first three steps were executed in this chapter's fixture. The remaining steps were inspected as configuration and expected operational procedure.

Atlas should attach an action to each signal before enabling a page. For pull errors, determine whether the image reference exists, credentials are valid, the registry is reachable, and the node has disk capacity. For start

errors, inspect port conflicts and engine errors. For a failed agent scrape, verify the endpoint and control-plane heartbeat independently. This prevents a dashboard name from dictating the diagnosis.

Before enabling the bundled rules, Atlas should check target-label alignment and inspect whether each producer is actually wired. A collector that remains zero because its controller never starts can make a rate-based alert appear quiet forever. Add a separate acceptance check for expected producer activity or configuration, and treat that as monitoring coverage evidence rather than manufacturing samples to make the graph look populated.

An alert also needs a recovery observation. Seeing the counter stop increasing is different from seeing a failed task replaced and healthy. Counters retain history after the incident ends, so the rate window must age out. A successful recovery should include workload state and runtime evidence, while the monitoring record shows when the condition stopped holding. The chapter does not invent that timeline from one synthetic increment.

Cleanup removes only test-local state. No host monitoring profile was launched, so there is no Prometheus volume or Grafana instance to delete. Future live evaluation should use owned infrastructure, a bounded duration, synthetic labels, and saved rule/query versions. Retain absent data, mismatched labels, and false positives as findings rather than hiding them behind a screenshot of a green panel.

Atlas's monitoring contract is now sharper: every displayed number has a producer, type, label set, aggregation, and operational interpretation. Gaps such as unwired collectors, proxy panels, and missing current-state gauges remain visible engineering work. That is how observability supports decisions without becoming a collection of reassuring names.

Exercises

1. Trace docker_operation_errors_total from a failed wrapped call to its collector, scrape target, alert, and dashboard panel. Mark which steps the lab executed.
2. Explain why duplicate accepted failed reports can inflate tasks_failed_total without duplicating task-failed events.
3. Rewrite the description of Task State Counts and Node Status Counts to match their actual queries. Identify the additional source needed for authoritative current counts.
4. Design a live alert acceptance drill for a short pull-failure burst, sustained failure, and absent agent scrape. Explain why a single synthetic counter increment cannot establish alert firing or recovery.

Worked answers are in Chapter 38 solutions, with emission and configuration claims in the ledger.

Chapter 39 – Performance and Scale Tests That Say What They Measure

Atlas wants to know whether the system can manage a thousand services. The repository contains a thousand-service test, but the number alone is not a capacity answer. A simulated running report is cheaper than pulling an image, starting a container, performing a healthcheck, or committing a database transaction. The first task is to identify what work is actually inside the measured interval.

This chapter teaches you to read benchmark inputs, preserve repeated measurements, distinguish fake-agent convergence from runtime startup, and use profiling without exposing an administrative endpoint. Atlas's result is a bounded benchmark record, a small simulated-load observation, and a scoped interpretation of the optional scale test. None is presented as a production throughput promise.

Fake-Agent Load and Real Runtime Capacity

The load tool constructs MemoryService, creates fake nodes, creates services, and reports assigned tasks running through direct control-plane calls. It does not start an HTTP server or an agent Runner. Its initial convergence loop includes service creation and synthetic running reports. The container identifiers are strings prefixed with fake, not observed Docker objects.

The simulation then runs until a duration deadline, occasionally changes node heartbeat state, and probabilistically reports running tasks failed. It sleeps one hundred milliseconds between ticks. SchedulerIterations counts simulation ticks, not invocations of the scheduler's Plan function. ReconciliationIterations counts initial service creations. Naming a summary field does not change the operations performed beneath it.

The configured duration begins after initial convergence, so total command runtime includes setup plus the requested simulation interval and any final work. Failure rate is a probability per eligible running task per tick, not failures per second or a measured production incident rate. Atlas should not compare runs with different tick counts as if they experienced identical failure exposure.

A fixed seed constrains pseudorandom choices, but it does not make elapsed timing deterministic. Generated object IDs, iteration order, scheduler activity in other experiments, and host contention can still affect the result. Preserve the seed alongside the exact source and flags. For a regression about a particular workload identity, save the concrete fixture rather than expecting one integer to reconstruct every external circumstance.

The actual bounded fake-load run used ten nodes, one hundred services, three replicas, failure probability 0.02, one second, and seed 42. It reported 323 task records, 260 running and 63 failed, with zero reported scheduler/reconciler errors, ten simulation ticks, and one hundred reconciliation iterations. These are observed output values from that run, not an expected transcript for every repeat.

Its average initial per-service convergence was 84.619 microseconds, and total initial convergence was 8.758617 milliseconds. Those intervals cover memory operations and synthetic reports. They do not include image transfer, container initialization, application readiness, persistent transactions, or network hops. The coexistence of failed historical tasks and replacement task records also means total task records should not be equated with desired live replicas.

Benchmark Inputs and Repeatable Results

The scheduler benchmark is more narrowly defined. It prepares one hundred ready nodes, each with eight CPU cores represented as eight thousand millicores and sixteen GiB allocatable memory. It prepares one

thousand pending tasks, each requesting one hundred millicores and sixty-four MiB, then repeatedly calls Plan on the same input. Setup occurs before ResetTimer.

```
        input := PlanInput{PendingTasks: tasks, Nodes: nodes,
Services: services}
        b.ResetTimer()
        for i := 0; i < b.N; i++ {
                _ = Plan(input)
        }
}
```

Exact source: internal/scheduler/benchmark_test.go:21–26.

This isolates planner computation and allocation behavior. The repeated input has no existing running-task inventory, no placement labels, and no published ports. A change that improves this benchmark can still regress a label-heavy or reservation-heavy case. Atlas should preserve representative input families rather than use one attractive number as the universal scheduler cost.

The actual bounded benchmark command selected only scheduler scoring and service/task list handlers, used a one-hundred-millisecond benchmark target, repeated three times, and set the benchmark CPU count to two. The record pins Go 1.25.0, Linux amd64, source SHA, UTC times, command flags, and raw output. The host identified its processor as AMD Ryzen 5 5500. Other concurrent work on this shared host was not controlled.

For scheduler scoring, the three measured results were:

Sample	Iterations	Time per Plan call	Allocated bytes per call	Allocations per call
1	9	11,991,457 ns	718,664	630
2	9	12,012,169 ns	718,664	630
3	9	12,050,878 ns	718,664	630

The median is 12,012,169 nanoseconds per operation, with the observed range shown by the samples. Three short samples do not justify a confidence interval, a tail-latency claim, or a host-independent benchmark. The summary artifact derives those figures mechanically from the raw scheduler lines and records its limitations. It intentionally does not attribute interleaved API logging lines to guessed benchmark names.

The HTTP list benchmarks use an in-process recorder and a prepared memory control plane containing one thousand services. They include handler processing and JSON output, but not a TCP exchange, TLS, a reverse proxy, or database reads. Default logging also contributes observable work. Comparing these with a network client benchmark would mix different costs unless the experimental design explicitly accounts for them.

Commit 7985b9ebe276f8447e1b8abfceb0d6b746ea6f2d added these benchmark and scale artifacts in a broader operations batch. Earlier 7fcdaf5877145b77bb484b2bc7cabae8eb75acab added the load tool. Their commit boundaries do not supply a performance history by themselves; measurements require comparable recorded runs.

Profiles and the Debug Boundary

A profile helps identify where measured work occurs; it does not create a trustworthy workload definition. Atlas can capture a scheduler CPU profile from the benchmark without opening a server listener. Use an owned output directory and retain the matching test binary so symbol interpretation corresponds to the captured program. A very short profile provides few samples and should be treated as a diagnostic sketch.

The chapter's profile capture succeeded against the same deterministic planner fixture. The downloaded toolchain lacked a prebuilt pprof executable, so the ordinary go tool invocation failed and that failure remains recorded. A source-built tool fallback is separately recorded; it must not be confused with a successful invocation of a missing binary.

The source-built fallback actually succeeded, as preserved in profile-top-source-tool.json. It reported a 301.20-millisecond capture with 120 milliseconds of CPU samples; map lookup and comparison work appeared in the short top listing. That observation suggests where to investigate the planner fixture, but the sampling interval is too small to declare a stable optimization priority. Keep the profile and matching binary identities together, and repeat a longer bounded capture before making a substantial change based on those percentages.

The application also offers an optional debug listener. The debug handler registers pprof routes only when enabled, and main starts it on a separate configured address. The default address is loopback. The handler itself has no authentication middleware, so a configuration change exposing it on a public interface changes the security boundary. Being separate from the public listener is useful but does not make every configured address safe.

Profiles can contain operational paths, command-line material, and memory-related data. For a real server procedure, keep access on an approved management path, bound the capture, and handle output as sensitive. This chapter did not enable that listener or collect a live service profile. Its fixture contains synthetic service/task identifiers and no production credentials.

The failure window for performance investigation is often the investigation itself. Enabling an unrestricted debug endpoint during an incident can expose information or add load. Atlas should prepare a reviewed profiling procedure before an outage and test that disabled public routes remain unavailable. Do not turn an urgent performance question into an unreviewed change to network access.

Interpreting a Thousand-Service Test

The optional scale test is behind the scale build tag. It runs fifty fake nodes, one thousand services, three replicas, a one-percent failure probability, one second of simulation, and seed 42. Its acceptance checks at least

three thousand task records and exactly one thousand initial reconciliation iterations. It does not assert that all desired replicas are healthy at the end.

The actual tagged execution passed. This establishes that the specified fake-memory scenario completed and met its two assertions in this environment. It does not establish one thousand real containers, PostgreSQL throughput, a production admission rate, or durable recovery under load. A database process running beside it would not alter that scope because the command never uses it as active state.

For Atlas's lab, use the disposable copy and the exact bounded commands in the records. Keep ordinary tests excluded from the benchmark command and separate the tagged scale invocation. Observe output and sample variation rather than rewriting expected numbers into a passing assertion. No host containers, published ports, or databases are created by these commands.

A production-shaped follow-up needs a different design. Define the number of services, replicas, images, image-cache state, update rate, placement constraints, event retention, database schema, network path, and readiness criterion. Measure end-to-end acceptance and convergence separately from planner CPU. Include errors, pending work, recovery after injected faults, and cleanup duration. Those dimensions are proposed requirements, not missing columns that can be filled by extrapolating this benchmark.

When comparing a candidate optimization, keep the old and new fixture definitions identical, run repeated samples under the same toolchain and CPU setting, and preserve rejected alternatives. A faster result obtained by removing capacity checks would violate correctness rather than improve it. The comparison must include the existing property and constraint tests, while the benchmark records the cost of preserving those semantics. Report both the time change and allocation change, and distinguish measured variation from a repeated directional result.

Resource cost also grows outside the hot function. Event and task history can expand while replicas remain constant; list endpoints may serialize large responses; retained failures can consume memory; agents and container logs add independent load. Atlas should observe steady state and bounded history as well as initial creation. The small chapter experiment intentionally avoids pretending that a brief creation burst covers those longer timelines.

The final performance statement is therefore specific: a pinned pure planner fixture produced three short observed samples; a bounded fake-memory load completed; an optional thousand-service simulation met its limited acceptance checks. These results help locate regressions and guide further experiments. They do not set a safe production cluster limit, and Atlas should retain that distinction in release notes and capacity decisions.

Exercises

1. Identify which operations are included in average_convergence_time and which real runtime costs are absent. Explain the actual meanings of SchedulerIterations and ReconciliationIterations.
2. Reproduce the scheduler benchmark with the recorded flags. Compare the sample range without inferring a production capacity or statistical confidence interval from three short runs.
3. Explain why a debug listener bound to a public address remains dangerous even though pprof is absent from the public API handler.
4. Design a production-shaped thousand-service acceptance experiment. Include authoritative readiness, durable state, runtime work, retained history, failures, and scoped cleanup rather than extrapolating the fake scenario.

Worked answers are in Chapter 39 solutions; classifications and measurement limits are in the claim ledger.

Chapter 40 – Release a System You Can Honestly Operate

Atlas has reached the end of the implementation tour with an operating local demonstration, a substantial set of packages, and several important limits. The release decision must account for all three. A successful workload demonstration does not erase memory-state loss on server restart, and an unwired durable store does not invalidate every useful scheduler or agent test. The task is to describe the system precisely enough that another operator can make a responsible decision.

This chapter assembles an operational dossier instead of a production-readiness slogan. You will connect actual test records with their prerequisites, read deployment manifests as conditional procedures, and interpret a database restore against the correct state owner. The final Atlas decision retains unresolved requirements and turns them into acceptance-oriented engineering work.

Release Checks With Honest Coverage

The repository's CI runs formatting checks, vet, OpenAPI tests, lint, coverage tests, and a separate race job. The workflow configures a PostgreSQL service for its quality job. That is a source-defined procedure. It is not evidence that this exact workflow completed in the edition's captured verification or that every integration fixture is compatible with the latest schema.

The actual book baseline built the four application binaries from the locked disposable source with Go 1.25.0. The successful build record follows earlier failed attempts caused by the inherited toolchain environment. Preserve both the diagnosis and corrected environment. A command record's exit status is a better witness than a filename suggesting that an export succeeded.

The baseline tests and race run passed their reached paths, retaining explicit Docker/PostgreSQL skips. Named test pass events must be distinguished from package summaries when counting results. The separate owned PostgreSQL fixture run failed two store tests because it applied only migrations one through fourteen while current queries require a column added in sixteen. That failure remains part of the dossier.

Later full-schema and guarded Docker extensions supply additional evidence, but they do not rewrite the original test outcomes. A full-schema store experiment is not the default server using PostgreSQL. A real-Docker harness with owned cleanup is not an unmodified Compose deployment. A race detector passing selected executions is not proof that all possible interleavings are safe. Release language must preserve these qualifications.

The Makefile lint target falls back to formatting checks and go vet when golangci-lint is absent. That fallback is useful, but it is not equivalent to running the configured linter. Record which tool actually ran. Similarly, parsing OpenAPI checks document structure; comparing routes checks inventory; cross-layer request/response tests check behavior. None should claim the others' coverage automatically.

Commit 7985b9ebe276f8447e1b8abfceb0d6b746ea6f2d added operational hardening and release material. The current release notes explicitly retain the memory-control-plane boundary. Atlas should keep that honesty in its own deployment report rather than replacing it with a checklist whose final green box overrides the architecture.

Deployment Manifests and Runtime Prerequisites

The production Compose file requests explicit server and agent image variables and requires several secrets. It creates PostgreSQL, server, agent, and optional monitoring services. The agent mounts the host Docker socket. These are starting points for an operator, not a managed

platform installation. They depend on trusted nodes, chosen image identities, network restrictions, and a reviewed secret-delivery mechanism.

The agent systemd unit joins the docker group and grants socket access. Host hardening options remain valuable, but Docker API authority can still amount to host control. Atlas should treat agent compromise as a node-compromise boundary, prevent workloads from receiving the socket, and restrict agent endpoints to appropriate networks. A non-root Unix username alone does not remove the authority granted by that socket.

The server unit runs as its own identity with a protected filesystem and environment file. Restart=always can restore process availability after failure, but it cannot persist MemoryService maps. A restart policy that makes the endpoint healthy again may therefore hide state loss unless the operational checks also compare expected services, tasks, credentials, and events. Process supervision and durable recovery are independent concerns.

The Compose server environment supplies DATABASE_URL and TLS placeholders. A database URL does not activate durable control-plane storage; TLS placeholders do not implement a native server TLS mode. The documented deployment assumes a reverse proxy or load balancer terminates TLS. Atlas must inspect actual network listeners and proxy configuration instead of assuming that mounted certificate files are being consumed.

Pin all deployment artifacts before a reproducible installation. Some optional monitoring images default to latest in the example. A future execution using those defaults can select different software from the reviewed run. Record digests or immutable tags, tool versions, configuration, and expected ports. This chapter does not fetch or launch those public images, install systemd units, create host users, or modify the machine's firewall.

Commit 18fcf32524007ba7d8a71acd70e5acc48f758e49 added the deployment examples. Their checked-in presence establishes a reviewable starting configuration. It does not establish that Atlas has a working TLS proxy, complete cross-node service networking, fenced stateful storage, or an approved production trust model.

A Backup Drill With the Correct State Boundary

The backup script requires an explicit database URL and creates a PostgreSQL custom-format dump without owner or privilege restoration. The restore script requires an explicit target, an existing input file, and acknowledgment. The guard is visible in source:

```
if [[ "$yes" != true ]]; then
  cat >&2 <<'WARNING'
Refusing to restore without --yes.

Restore modifies the target database. If --clean is also supplied,
existing
objects in the target database may be dropped before being
recreated.
WARNING
  exit 2
fi
```

Exact source: scripts/restore-db.sh:73–81.

Acknowledgment protects against accidental invocation; it does not inspect whether the selected target is safe. Atlas should restore into a newly owned disposable database, retain its exact name, verify that it was absent before creation, and remove only that target afterward. Never demonstrate destructive clean against a live database merely to show that the script accepts the flag.

The backup runbook distinguishes database rows from external configuration and runtime objects. A dump excludes Docker images, containers, volumes, process memory, TLS material, JWT signing secrets, bootstrap tokens, and the local envelope key. Encrypted secret rows without the

matching envelope key remain rows that cannot yield their original plaintext. Backup success therefore needs both data verification and a separate secure key-recovery plan.

The actual full-schema recovery record created owned source and restore databases, applied all sixteen migrations, inserted scoped records and synthetic encrypted canary data, dumped and restored, and verified the restored values. It rejected a wrong decryption key. It also retained migration idempotence and competing task-claim observations. Its program is a teaching extension around the store package, not an alternative executable silently substituted for orch-server.

That drill establishes useful package-level durability and restore behavior. It cannot restore active records created only inside a default MemoryService, because those records were never part of the database dump. The separate actual fresh-server experiment observed exactly that distinction: the second process was healthy enough to answer HTTP while its prior services, tasks, and nodes were absent. The surviving captured Docker container belonged to another state owner.

A recovery procedure should therefore specify what it intends to reconstruct. Database restoration recovers selected persisted records. Agent reconciliation inspects managed engine objects. Image registries supply images. Volume restoration recovers workload data. Configuration/key recovery restores identities and decryption capability. A single successful pg_restore command cannot certify all of those owners or their coordinated consistency.

Commit 64abe001e83932cba294831424557dfd16f77c8c added backup scripts and guidance. The source remains useful when its conditional persistence premise is retained. Atlas's release dossier must say whether it is evaluating the store package, a proposed durable composition, or the actual default binary. Mixing those modes is the most consequential backup error in this case study.

The Final Readiness Decision and Backlog

The dossier verifier checks selected evidence records exist, match the locked source, and retain their observed success status and output files. Its actual corrected execution passed. The initial inventory deliberately failed on an incorrectly selected failed build record; the corrected inventory points to the actual successful build. Both records remain visible.

Passing this verifier means the evidence inventory is internally connected. It explicitly reports production_decision as not approved. It does not certify application safety, replace independent review, or rerun every experiment. This distinction prevents the final book check from becoming an accidental production approval mechanism. The result is a reviewable local dossier, not a deployment action.

Requirement	Evidence today	Acceptance work still needed
Durable active state	Store, migrations, owned restore exercise	Compose a transactionally equivalent durable server
Exclusive controllers	Lease library and loss tests	Wire loops, repair session cleanup, verify topology and handover
Resource enforcement	Direct adapter works; normal path drops limits	Preserve requested fields through DTO and agent into engine
Autoscaling	Controller/provider interfaces and deterministic package tests	Supply observed metrics and explicit server composition
Node trust	Enrollment and credential policy mechanisms	Complete transport identity and operational rotation model
Networking/ stateful recovery	Local and documented boundaries	Cross-node connectivity, fencing, and volume-aware recovery

These backlog items should be ordered by the guarantee Atlas intends to offer. Durable active state precedes credible fresh-server recovery. Leadership composition follows shared authoritative state and must resolve retained advisory locks. Runtime resource propagation is a

separate acceptance gate even for a single server. Complete node identity and networking cannot be deferred behind a claim that workloads are securely isolated across hosts.

Each improvement needs a negative case as well as a happy path. Durable transactions should survive an injected failure without partial service/event writes. Leadership should reject wrong cluster identity and stop mutating work after loss. Resource propagation should preserve explicit values and reject malformed inputs. Metrics providers should distinguish unavailable observations from low utilization. Networking tests should measure actual reachability rather than replay a diagram.

The Atlas case study demonstrated real guarded deployment, logs, scale, deletion, resource-boundary differences, actual memory-state loss after process restart, package-level database restoration, deterministic controller decisions, and scoped performance observations. The dossier also preserves harness corrections and explicit skips. That combination is more useful than either declaring everything ready or discarding every working component because production prerequisites remain incomplete.

An operator can now make a bounded decision: use the current system for the documented local learning and demonstration scenarios, with owned resources and understood failure boundaries; require further implementation and evidence before relying on durable production orchestration. The book ends with that concrete decision because an honest operating contract is part of the engineering, not a qualification added after the release is advertised.

Exercises

1. Build a coverage table separating the ungated suite, explicit integration skips, original PostgreSQL fixture failure, full-schema extension, and guarded real-Docker flow.
2. Review the systemd restart policy and Compose database/TLS variables. Explain why they do not establish durable state or native TLS consumption.

3. Describe the owned restore drill, including database identity, canary verification, key handling, and cleanup. Identify the state a dump cannot recover from the default server.
4. Run the dossier verifier and select the next Atlas guarantee to implement. Define its failure case, evidence, and rejection criterion without changing production_decision merely because an inventory check passes.

Worked answers are in Chapter 40 solutions, with source and dossier classifications in the claim ledger.

Appendix A – CLI, API, and Configuration Reference

A command name is an interface, not an operational guarantee. This reference describes the pinned v0.3.0 implementation and distinguishes locally executable commands from requests that need a running server. The machine-readable companion records all 61 explicitly constructed Cobra nodes, 75 registered HTTP method/path pairs, 49 top-level and nested configuration fields, their exact source provenance, and the comparison with OpenAPI. The constructor inventory replaces the machine-specific CLI configuration path with the explicit `REFERENCE_CONFIG.yml` option; this privacy override is not the production default path. Server and agent defaults are captured before the separate precedence example sets its environment override. Automatic help and completion nodes are excluded from the authored-command count; Cobra creates those during execution.

CLI lookup

The five persistent options are `--server`, `--token`, `--namespace`/`-n`, `--output`, and `--config`. Output accepts `table` or `json`; an unsupported value is rejected before a command runs. This does not make every command a tabular resource listing: validation, version printing and delete acknowledgments have their own output. Usage export additionally supports its own `--format` selection.

The CLI has **no built-in server URL**. This differs from the agent's default connection address. Local `version`, `validate`, and generated help do not need a connection. Commands that construct an API client require a server URL from flags, environment or configuration. A group node such as `orch service` is a navigation parent, not an independent service operation.

The table records explicit source commands. “No validator” means Cobra does not enforce argument arity at that node; it does not authorize adding undocumented operands. Exact flag types, defaults, shorthands and accepted argument-count probes are retained in the companion inventory instead of being guessed from usage text.

Command	Operands	Local options
`orch audit`	`none`	`--action`, `--actor-id`, `--actor-type`, `--limit`, `--outcome`, `--since`, `--target-id`, `--target-type`
`orch cluster`	`none`	inherited only
`orch cluster check-upgrade`	`none`	inherited only
`orch cronjob`	`none`	inherited only
`orch cronjob apply`	`<file.yaml>`	inherited only
`orch cronjob ls`	`none`	inherited only
`orch cronjob resume`	`<id>`	inherited only
`orch cronjob suspend`	`<id>`	inherited only
`orch delete`	`<service-name-or-id>`	inherited only
`orch deploy`	`<file.yaml>`	inherited only
`orch endpoints`	`<service-name-or-id>`	`--include-unhealthy`
`orch events`	`none`	`--follow`, `--service`, `--severity`, `--type`
`orch gitops`	`none`	inherited only
`orch gitops add`	`none`	`--branch`, `--drift-policy`, `--interval`, `--path`, `--prune`, `--repo`
`orch gitops delete`	`<id>`	inherited only
`orch gitops diff`	`<service>`	inherited only

Command	Operands	Local options
`orch gitops ls`	`none`	inherited only
`orch gitops status`	`none`	inherited only
`orch gitops sync`	`<id>`	inherited only
`orch job`	`none`	inherited only
`orch job logs`	`<job>`	inherited only
`orch job ls`	`none`	inherited only
`orch job run`	`<file.yaml>`	inherited only
`orch logs`	`<service-name-or-id>`	`--follow`, `--tail`, `--task`
`orch maintenance`	`none`	inherited only
`orch maintenance create`	`<name>`	`--duration`, `--enabled`, `--global`, `--operations`, `--schedule`, `--timezone`
`orch maintenance delete`	`<id>`	inherited only
`orch maintenance ls`	`none`	inherited only
`orch namespace`	`none`	inherited only
`orch namespace create`	`<name>`	inherited only
`orch namespace delete`	`<name>`	inherited only
`orch namespace ls`	`none`	inherited only
`orch node`	`none`	inherited only
`orch node drain`	`<node-id>`	`--force`
`orch node drain-status`	`<node-id>`	inherited only
`orch node inspect`	`<node-id>`	inherited only
`orch node ls`	`none`	inherited only
`orch node uncordon`	`<node-id>`	inherited only
`orch quota`	`none`	inherited only

Command	Operands	Local options
`orch quota get`	`none`	inherited only
`orch quota set`	`none`	`--max-cpu-millicores`, `--max-memory-bytes`, `--max-public-ports`, `--max-registry-credentials`, `--max-replicas`, `--max-secrets`, `--max-services`
`orch retention`	`none`	inherited only
`orch retention prune`	`none`	`--dry-run`
`orch retention status`	`none`	inherited only
`orch rollback`	`<service-name-or-id>`	`--force`
`orch rollout`	`<service-name-or-id>`	`--force`, `--image`, `--max-surge`, `--max-unavailable`
`orch rollout status`	`<service-name-or-id>`	inherited only
`orch scale`	`<service-name-or-id>`	`--force`, `--replicas`
`orch service`	`none`	inherited only
`orch service inspect`	`<service-name-or-id>`	inherited only
`orch service ls`	`none`	inherited only
`orch service ps`	`<service-name-or-id>`	inherited only
`orch usage`	`none`	`--from`, `--to`
`orch usage export`	`none`	`--format`
`orch validate`	`<file.yaml>`	inherited only
`orch version`	`none`	inherited only
`orch volume`	`none`	inherited only

Command	Operands	Local options
`orch volume create`	`<name>`	`--node`
`orch volume inspect`	`<id>`	inherited only
`orch volume ls`	`none`	inherited only

`orch-server migrate status|up|down` is dispatched by the server entrypoint before ordinary server flags are parsed. It is not an `orch migrate` Cobra command. Migration options include `--database-url` and `--allow-down`; status and up are schema operations, while down requires explicit permission. Execute schema changes only against the disposable database defined in a laboratory. Merely printing help must never perform a migration.

Configuration precedence without a connection

For CLI connection settings, precedence is nonempty explicit flag, nonempty environment variable, YAML file, then the empty/default value. Namespace falls back to `default`. This fully specified example demonstrates the addresses without sending traffic:

```
reference_dir=$(mktemp -d /tmp/orch-reference.XXXXXX)
cat > "$reference_dir/cli.yaml" <<'YAML'
server_url: http://127.0.0.1:18079
namespace: atlas-reference
YAML
ORCH_SERVER_URL=http://127.0.0.1:18080 \
  orch --config "$reference_dir/cli.yaml" \
  --server http://127.0.0.1:18082 version
```

Version printing is local and therefore does not itself reveal the selected URL. The companion offline constructor harness separately calls `config.LoadCLI` with the same file/environment/override values and observes `http://127.0.0.1:18082`, namespace `atlas-reference`. This is an executed precedence check, not a successful connection to that address. All three addresses are deliberate loopback examples. Remove only the temporary directory created by this example after retaining any evidence you need.

Server and agent configuration follow defaults, YAML, environment, then explicit overrides. Blank strings generally do not erase earlier nonempty settings. Boolean pointer overrides preserve an explicit false; do not generalize string behavior to every field. File configuration is decoded through dedicated structs and YAML unmarshalling rather than the stricter service-manifest decoder. A misspelled config key therefore needs an explicit review, even if loading succeeds.

Configuration family	Loaded/ validated	Actual consumer and boundary
Server address and shutdown TTL	Address required; TTL positive	HTTP listener and bounded shutdown
Database URL	Required nonempty	Migration runner and optional schema check; does not select live persistence
JWT secret/users/ bootstrap token	Loaded; principal parsing and handler checks	Optional user JWT gate, static role override, agent enrollment
Secret key	Nonempty required	Local envelope provider; loss prevents decrypting retained ciphertext
Heartbeat timeout/monitor interval	Positive durations	Node monitor; expiry is suspicion, not fencing
Debug address/pprof	Address required when enabled	Separate debug listener; profiling defaults off
Retention	Durations must not be negative	Memory history pruning; zero passes validation despite “positive” error wording
Cluster policy	Maximums must not be negative	Admission decisions; field propagation must still be checked
Agent name/address/labels	Required name and addresses	Registration, heartbeat and log listener identity
Agent Docker socket path	Loaded and required	**Ignored by runtime construction**; Docker SDK environment chooses its transport

Configuration family	Loaded/ validated	Actual consumer and boundary
CLI server/token/ namespace	Selected by precedence	API client URL, bearer header and namespace header

The inventory identifies each remaining field individually, including nested policy switches and retention durations. Sensitive fields are replaced wholesale in `orch-server config print --config <dedicated-file>` and `orch-agent config print --config <dedicated-file>`. These are actual executable options. Redaction removes database URLs, token/key material and user maps rather than merely hiding a password substring. It does not sanitize arbitrary workload logs, terminal history or all error messages.

`DATABASE_URL` reaches the server migration dispatcher; the ordinary configuration loader reads that same variable. `ORCH_SCHEMA_CHECK=true` opens the configured database to validate its schema before serving. After that check, the executable still constructs `controlplane.NewMemoryService`. A valid schema and a green startup cannot make desired services, node credentials or operations survive process restart.

HTTP boundaries

All public workload requests select `X-Orch-Namespace` before the `namespace` query value; normalization supplies the default namespace. This is selection, not authorization. User JWT enforcement is conditional on a nonempty signing secret. With that configured, viewers read the older resource surfaces, operators perform the explicitly classified service mutations, and administrators own namespace/quota and newer fallback routes. In particular, the job, cron, volume, notification and operation surfaces fall through to admin rather than inheriting a guessed operator role.

The four `/v1/agent/` operations form a different trust surface. Registration exchanges bootstrap authority for an issued node credential. Heartbeat and assignment/status operations validate that credential and

the relevant node identity; task reports additionally require assignment ownership and valid desired/actual transitions. Public OpenAPI deliberately omits these four agent routes.

Requests propagate or generate `X-Request-ID`; normal application errors contain an error code, message and request ID, with redacted details when supplied. Typical mappings include not found, invalid argument, conflict, quota exceeded, unauthorized, forbidden, failed precondition, unavailable and internal error. Inspect the route-specific response symbol and status rather than expecting every success to be 200: creates use 201, asynchronous rollout/rollback uses 202, and deletes commonly use 204. Raw mux errors and established streaming responses can differ from the JSON envelope.

Deletion of a service is soft: its lifecycle changes to deleting, and cleanup must reach removed tasks before the service becomes deleted. An accepted request is not proof of container disappearance. Operational conflicts such as an incompatible concurrent rollout return a conflict rather than silently replacing the active operation. `/healthz` and `/readyz` report process status without querying a durable backing store.

Units and discrepancy ledger

JSON API DTOs use integer CPU millicores and memory bytes. Deployment YAML accepts human-readable CPU/memory quantities and converts its single resource pair into both requests and limits. It is not a separate arbitrary requests/limits hierarchy. Durations in domain JSON use Go `time.Duration` integer nanoseconds, while deployment YAML and file configuration accept textual duration formats through their respective parsers. Configuration's fallback integer parser treats an integer string as seconds; do not reuse that rule for JSON DTO integers.

The static OpenAPI and registered user handlers have four coverage differences: `/metrics`, `/v1/discovery/services`, `/v1/discovery/services/{name}`, and `/v1/integrations/traefik/config` are implemented but absent from the specification. Every documented

method/path has a registered handler at this revision. Parameter-name normalization was used only for comparison; the companion preserves the actual path names and wildcard suffix.

Documentation advertises a Docker socket flag; the field loads and validates, but the agent constructor calls `NewEngineRuntimeFromEnv` without it. A reader changing only that flag can target the wrong engine. Documentation's volume overview refers broadly to claims, while the CLI creates/inspects volumes and the API separately exposes claims. Neither wording creates an unimplemented CLI claim command. These discrepancies remain visible rather than being corrected in the immutable application snapshot.

Source anchors: Cobra tree, workload commands, configuration, registered handlers and middleware, server composition, agent runtime construction, OpenAPI, and deployment parser. Generated help for all 61 authored command nodes, one complete Atlas service manifest, and both redacted config-print paths were also executed locally with zero exits. None contacts an API or Docker engine. This appendix is a source reference plus scoped offline execution, not a connected-cluster certification.

Source-backed boundary statements for this appendix are indexed in the appendix claim record. Detailed command, state, exercise and reference tables retain their own source provenance.

Appendix B – Lifecycle States and Invariant Reference

A lifecycle reference answers two separate questions: whether an edge is valid, and who is allowed to request it. The server owns desired task state. Assigned agents report actual state through an authorization and transition boundary. Node monitors and policy logic can also change task records for explicit reasons. A state name alone never grants ownership, and a condition such as `node_lost` is not a replacement state machine.

The tables below were generated by actually invoking the pinned public lifecycle validators in a disposable source copy. They include same-state acceptance separately, making repeated observations visible without pretending they are new transitions. Their confidence is the local helper contract: constructor execution and finite state checks do not prove a distributed execution will converge under every failure.

Allowed transitions

Service

From	Different allowed destinations	Duplicate
`active`	`deleting`	accepted
`deleted`	none	accepted
`deleting`	`deleted`	accepted

Task actual

From	Different allowed destinations	Duplicate
`assigned`	`pulling`, `created`, `running`, `healthy`, `unhealthy`, `stopping`, `stopped`, `removed`, `failed`	accepted

From	Different allowed destinations	Duplicate
`created`	`starting`, `running`, `stopping`, `stopped`, `removed`, `failed`	accepted
`failed`	`removed`	accepted
`healthy`	`running`, `unhealthy`, `stopping`, `stopped`, `removed`, `failed`	accepted
`pending`	`assigned`, `stopped`, `removed`, `failed`	accepted
`pulling`	`created`, `running`, `stopping`, `stopped`, `removed`, `failed`	accepted
`removed`	none	accepted
`running`	`healthy`, `unhealthy`, `stopping`, `stopped`, `removed`, `failed`	accepted
`starting`	`running`, `stopping`, `stopped`, `removed`, `failed`	accepted
`stopped`	`removed`	accepted
`stopping`	`stopped`, `removed`, `failed`	accepted
`unhealthy`	`running`, `healthy`, `stopping`, `stopped`, `removed`, `failed`	accepted

Task desired

From	Different allowed destinations	Duplicate
`assigned`	none	accepted
`created`	none	accepted
`failed`	none	accepted
`healthy`	none	accepted
`pending`	none	accepted
`pulling`	none	accepted
`removed`	none	accepted

From	Different allowed destinations	Duplicate
`running`	`stopped`, `removed`	accepted
`starting`	none	accepted
`stopped`	`removed`	accepted
`stopping`	none	accepted
`unhealthy`	none	accepted

Node

From	Different allowed destinations	Duplicate
`draining`	`ready`, `offline`	accepted
`offline`	`ready`	accepted
`ready`	`draining`, `offline`	accepted
`unknown`	`ready`, `offline`	accepted

Deployment

From	Different allowed destinations	Duplicate
`failed`	none	accepted
`paused`	`running`, `failed`	accepted
`pending`	`running`, `succeeded`, `failed`, `rolling_back`, `rolled_back`	accepted
`rolled_back`	none	accepted
`rolling_back`	`paused`, `failed`, `rolled_back`	accepted
`running`	`paused`, `succeeded`, `failed`	accepted
`succeeded`	none	accepted

Desired-state construction normally uses `running`, `stopped` and `removed`. The generic desired validator first accepts any identical known TaskStatus values; therefore its matrix includes identity edges for other enum values. That is helper behavior, not permission for an agent to introduce those desired states. Starting and stopping are also recognized actual enums, but the ordinary agent report whitelist does not admit them. Their presence reserves lifecycle vocabulary without inventing a runtime progress feature.

Service deleted is terminal: restoring an old deleted ID to active is forbidden. Task failed and stopped are terminal for execution, but the generic actual-state validator permits cleanup to removed. Removed cannot leave itself. Deployment succeeded, failed and rolled_back are immutable terminal histories; rollback creates another deployment with its own ID and rolling_back state. Paused is accepted by the enum/validator but lacks an operator pause/resume CLI feature at this revision.

The agent guard narrows the generic task matrix. An actual failed task with desired running cannot report removed merely because ValidateTaskTransition permits that edge; AgentCanReportTaskStatus accepts only a duplicate terminal report in that branch. Once desired state is stopped or removed, the cleanup branch permits stopped, removed or failed. An already removed task still accepts only removed. This order matters when interpreting a late worker message.

Jobs and cron are separate mechanisms

Jobs have pending, running, succeeded and failed states. They do not call a generic ValidateJobTransition helper: MemoryService derives their state from associated task reports. Running or healthy work marks a job running. A stopped task uses an explicit exit code when available and otherwise zero; zero means succeeded. A failed task defaults its missing exit code to one. Retry budget checks compare Attempts to BackoffLimit before increasing attempts and creating a new pending task identity.

These rules must not be inferred from the service restart state machine. A retry creates another task; it does not resurrect a terminal task in place. Cron records scheduling metadata such as LastScheduleAt and LastJobID rather than a lifecycle enum equivalent to deployment status. Allow, Forbid and Replace govern overlapping jobs; occurrence selection, catch-up and clock behavior belong to the cron controller. A suspended cron entry does not imply its already running job was canceled.

Invariant index and scope

ID	Invariant	Enforcing boundary	Counterexample outside scope
INV-01	Replica convergence	`reconcileServiceTasksLocked`	Zero capacity or restart never can leave intent unfulfilled; initial immediate placement bypasses richer Plan constraints.
INV-02	Atomic task claim	`AssignTask`	One-winner claim does not establish default server persistence or remote process fencing.
INV-03	State-change event ordering	`ReportTaskStatus`	Memory mutation and append are not a durable transaction across process failure; do not generalize to every controller append.
INV-04	Assigned node owns reports	`ReportTaskStatus`	Credential validity alone cannot authorize a different node's task; physical old-worker activity is not fenced.
INV-05	Namespace secrecy	`GetService`	A global identifier must not be returned before scope validation; inspect every newer workload surface separately.
INV-06	Rollout availability budget	`reconcileDeployment`	Running new tasks alone do not satisfy newHealthy; concurrent physical failure can invalidate a snapshot budget.

ID	Invariant	Enforcing boundary	Counterexample outside scope
INV-07	Owned container cleanup	`removeAssignedTask`	Server deleting status is not proof removal happened; cleanup can fail or old agent can remain unreachable.

Each companion invariant records assumptions, a precise source symbol and either a real test name or an explicit test gap. An inspected test is not labeled as a new run. Replica convergence requires an active service, an applicable restart policy, capacity and eventually successful operations. Atomic claim safety establishes one successful assignment transition for one pending record; it does not fence a worker that lost contact after starting a process.

Event emission follows validated state changes in the memory handler, and duplicate terminal reports avoid duplicate state-change events. This is not durable atomicity between a database commit and an external event stream. The store-backed reconciler has different transaction/error windows and requires its own constructor and backing store. Default process restart destroys MemoryService intent even if a PostgreSQL package test previously passed.

Namespace secrecy is a lookup and return-value property, not a naming convention. An implementation must verify context before returning data associated with a guessed identifier. Safe cleanup similarly depends on managed, node and task labels, not just a user-provided container name. Neither invariant should be extended to new handlers or runtime adapters without tracing those checks again.

Three source-derived traces

Trace 1: a repeated healthy report. Preconditions: the task belongs to the authenticated node, DesiredStatus is running, ActualStatus is healthy, and its service restart policy need not act. (1) The agent submits healthy again. (2) Node/task ownership is verified. (3) The guard accepts the identity edge. (4) The control plane retains healthy without creating another state-change event. A new health observation may carry

metadata; it must not be counted as a new lifecycle transition. The companion validator harness observed guard acceptance; full handler event behavior is established by the cited source test.

Trace 2: deletion during an active rollout. Preconditions: an active service has an in-progress deployment and running tasks. (1) DeleteService marks the service deleting. (2) Desired task state changes to stopped. (3) A late healthy report is rejected by the cleanup-intent guard. (4) The assigned agent stops and removes its owned container, then reports removed. (5) Finalization requires every associated actual task to be removed before the service is deleted. A previously captured rollout snapshot cannot make that service active again. Offline workers can leave deleting visible indefinitely; failed or stopped alone does not satisfy the finalization predicate. This is a reconstructed path, not a new Docker observation.

Trace 3: an offline stateless versus stateful task. Preconditions: two services differ by the Stateful flag and both have tasks on a node whose nonzero heartbeat is older than the monitor cutoff. (1) The node becomes offline. (2) Tasks receive a node_lost condition independently of their status. (3) Stateless work is failed with desired removed so replacement intent can be created elsewhere. (4) Stateful work retains assignment for manual recovery rather than pretending local data moved. (5) A later authenticated worker response must still satisfy current ownership and cleanup rules. Neither branch physically stops a partitioned process; there is no remote fencing token enforced by Docker. This trace describes source behavior and explicitly leaves split-brain prevention unproven.

Using the matrix in reviews

When adding a state, enumerate its owner, outgoing edges, duplicate behavior, cleanup path and every consumer switch. Start with an invalid representative edge such as failed to healthy, then inspect whether a higher-level policy replaces the task instead. Review stale reports against

both current desired state and authenticated assignment. A valid generic edge can remain forbidden at the agent boundary, and a valid store transition can remain unused by the shipped binary.

The matrix is a finite reference for finding missing cases, not a formal proof of distributed correctness. Its most useful review question is concrete: after this message is delayed, retried or delivered to a restarted process, which identity and owner still exist to authorize it?

Source: lifecycle validators and report guard, transition tests, MemoryService transitions, agent ownership/cleanup tests, job bookkeeping, rollout controller, and store claim tests.

Source-backed boundary statements for this appendix are indexed in the appendix claim record. Detailed command, state, exercise and reference tables retain their own source provenance.

Appendix C – Commit Atlas and Original Prompt Crosswalk

History is useful when it explains how a mechanism changed and what remained unresolved. It becomes misleading when a prompt title, a tracker mark or a release heading is substituted for executable evidence. This atlas keeps implementation, documentation and the original authoring requests separate while making their connections inspectable.

The pinned repository has **105 reachable commits and 91 original numbered prompts**. These counts were independently verified from a separate read-only bare Git clone, against the locked full SHA and tree object. Reachable means every ancestor, including merged side parents; the narrower first-parent count is not an interchangeable total. The book's own 000–099 workflow is a different prompt sequence and is not included in the original 00–90 inventory.

The companion history contains every full commit SHA exactly once, its parents, authored and committed timestamps, subject, changed paths and immutable link. Every original prompt appears once with its complete filename, pinned blob URL, earliest containing addition commit, historical blob object ID and thematic chapter destinations. No author email addresses are copied into this reference.

Reading the crosswalk honestly

Original prompt files entered the repository in two containing commits: 9f94bdad8bcf9f5bb819ec0b711440467a15da9b, which introduced thirty files, and f0c526fafac58e7921e32ac314f594f7fe14b8b3, which introduced sixty-one. First-containing means a Git addition was found and its blob presence checked; it does not mean the requested feature was first implemented then. In particular, June implementation can predate July prompt publication.

The crosswalk uses four evidence grades. A direct textual match requires explicit text connecting the records; it is not awarded merely because a title resembles another title. Strong path/diff correspondence means the associated implementation changed the named mechanism and the historical path exists. Partial relationship covers design/request material whose full executable delivery is not established. Unresolved records retain a specific missing evidence question instead of silently becoming support. The current mappings use strong correspondence or partial relationships; none is promoted to a literal execution transcript.

Associations can be many-to-one or one-to-many. The late quota/GitOps, jobs/storage and operations batches deliberately serve several prompts. Earlier docs and later corrections remain their own history entries even when they do not map neatly to an implementation request. Timestamps provide sequence context, not proof of the author's private intent.

Twelve checkpoints

Each excerpt below was retrieved from the named historical Git blob, with its exact bytes, line interval and checksum recorded. Short fragments illustrate the interface at that time; they are not complete replacement listings. Paths refer to that commit even when later code reorganized them.

1. Bootstrap: an executable shell

The scaffold introduced Go entrypoints, configuration validation and HTTP listener construction. Read the initial server run function before inspecting later controllers: its existence establishes a process skeleton, not the complete architecture promised in conventions. Later API construction selected an in-memory implementation. The enduring question is which constructor actually owns desired state.

Historical excerpt: cmd/orch-server/main.go at 72f27ed1d81b.

```
func run(ctx context.Context, logger *slog.Logger, cfg
config.ServerConfig) error {
        if err := cfg.Validate(); err != nil {
                return err
        }

        server := &http.Server{
```

2. Domain contracts: names become obligations

ServiceSpec collected typed workload inputs rather than leaving deployment intent as an unstructured map. Validation tests made malformed names, images and resources reviewable. Later versions expanded namespace, policy, security and workload fields. A field appearing in a struct is the beginning of its support trace; only its eventual runtime consumer establishes an end-to-end feature.

Historical excerpt: pkg/types/types.go at c756f9d7f3a4.

```
type ServiceSpec struct {
        Name                string                `json:"name"`
        Image               string                `json:"image"`
        Replicas            int                   `json:"replicas"`
        Env                 map[string]string
`json:"env,omitempty"`
        SecretRefs          []SecretRef
`json:"secret_refs,omitempty"`
```

3. PostgreSQL: a real package with a composition gap

The persistence commit added pgx-backed operations and integration tests. Its historical constructor accepts a pool explicitly, showing what a caller must supply. Subsequent migrations added optimistic concurrency, operational fields and attachment records. The default server never adopted this store as its live state owner; successful database package experiments therefore retain a narrower scope than a durable cluster.

Historical excerpt: internal/store/postgres.go at 7eba6544ffe2.

```
func NewPostgresStore(pool *pgxpool.Pool) *PostgresStore {
        return &PostgresStore{pool: pool}
}

func ConnectPostgres(ctx context.Context, databaseURL string)
(*pgxpool.Pool, error) {
        pool, err := pgxpool.New(ctx, databaseURL)
```

4. Agents: processes acquire an identity

Registration and heartbeat gave a worker a control-plane relationship and introduced a runner loop. Later issued credentials replaced using bootstrap authority for every control-plane call. Local task reconciliation, label ownership and drift cleanup refined execution safety. An enrollment mechanism can authenticate a worker without fencing a partitioned predecessor or protecting its Docker socket from local compromise.

Historical excerpt: internal/agent/agent.go at 5cf921384916.

```
func (r *Runner) Run(ctx context.Context) error {
        if err := r.cfg.Validate(); err != nil {
                return err
        }
        if r.client == nil {
                client, err := NewHTTPClient(r.cfg.ServerURL,
r.cfg.BootstrapToken)
```

5. Scheduler: planning meets persistence

RunOnce introduced a repeatable placement seam and tests around selection. Later property checks, resource accounting, public-port reservations and atomic claims expanded the contract. Follow both planning and write paths: ordinary MemoryService immediate placement still bypasses richer scheduler Plan constraints. A scheduler package's correct algorithm does not imply every creation path passes through it.

Historical excerpt: internal/scheduler/scheduler.go at fce0a657651c.

```
func (s *Scheduler) RunOnce(ctx context.Context) ([]Assignment,
error) {
        if s.store == nil {
                return nil, fmt.Errorf("scheduler store is
required")
        }
        if err := ctx.Err(); err != nil {
                return nil, err
```

6. Reconciliation: desired state becomes repeated work

The historical reconciler exposes ReconcileOnce and a store contract, making retryable convergence inspectable. Deletion and health refinements later changed which tasks count as active and when replacement is allowed. This package can reason about restarting from stored intent when explicitly wired to a durable store. The shipped memory binary instead uses its own integrated bookkeeping and loses live intent on process restart.

Historical excerpt: internal/reconciler/reconciler.go at 208eb99758a5.

```
func (r *Reconciler) ReconcileOnce(ctx context.Context) error {
        if r.store == nil {
                return fmt.Errorf("reconciler store is required")
        }
        if err := ctx.Err(); err != nil {
                return err
```

7. Rollout: availability becomes a budget

The rollout controller added repeated deployment processing rather than treating an image update as a single synchronous mutation. Tests exercise surge and replacement sequencing; later rollback and operation-lock changes preserve terminal history and reject incompatible overlaps. Current old-task availability permits running or

healthy, while new-version success specifically counts healthy. Read those predicates rather than translating every “running” observation into rollout success.

Historical excerpt: internal/rollout/rollout.go at de5900bced92.

```
func (c *Controller) RunOnce(ctx context.Context) error {
        if c.store == nil {
                return fmt.Errorf("rollout store is required")
        }
        deployments, err := c.activeDeployments(ctx)
        if err != nil {
```

8. Node trust: credentials have a lifecycle

The issued-agent credential constructor defaults its TTL to fifteen minutes, then stores token hashes and validates expiry. Heartbeat rotation and revocation tests refine trust beyond the original bootstrap exchange. A rotation response can be lost after the server replaces its stored hash, leaving a worker with an obsolete credential. This is an engineering failure window, not evidence that enrollment is permanently reliable.

Historical excerpt: internal/auth/agent.go at 4f25e78c229b.

```
func NewTokenAgentCredentialIssuer(ttl time.Duration)
*TokenAgentCredentialIssuer {
        if ttl <= 0 {
                ttl = 15 * time.Minute
        }
        return &TokenAgentCredentialIssuer{ttl: func() time.Duration
{ return ttl }}
}
```

9. Namespaces, quotas and GitOps: a shared change batch

The quota/image/GitOps batch implements several thematic prompts together. GitOps receives a store and parser, checks a source out and applies bounded manifests; quota accounting operates within namespace mutation paths. Presenting that commit as one prompt’s

exclusive result would erase the batch. Later job and operational surfaces require separate authorization checks, and digest syntax validation still does not resolve a registry's remote digest.

Historical excerpt: internal/gitops/gitops.go at f2836eabfff8.

```
func (controller *Controller) Sync(ctx context.Context, id string)
(types.GitOpsSource, error) {
        source, err := controller.store.GetGitOpsSource(ctx, id)
        if err != nil {
                return types.GitOpsSource{}, err
        }
        if err := ValidateSource(source); err != nil {
```

10. Finite and scheduled work: task semantics get another owner

The jobs/storage/DNS/notification batch adds CreateJob validation and task-derived bookkeeping. Cron selection, concurrency policy and occurrence records then coordinate finite work rather than ordinary service replicas. Job command/env/security fields require tracing to the agent; the Task-only dispatch path does not automatically consume every JobSpec field. Local volumes and DNS do not create replicated storage or an overlay network.

Historical excerpt: internal/controlplane/workloads.go at 4d641bbe8913.

```
func (s *MemoryService) CreateJob(ctx context.Context, spec
types.JobSpec) (types.Job, error) {
        if err := spec.Validate(); err != nil {
                return types.Job{}, fmt.Errorf("%w: %v",
store.ErrInvalidState, err)
        }
        req, err :=
spec.ResourceRequirements.WithDefaults(types.DefaultResourceDefaults
())
        if err != nil {
```

11. Operational hardening: useful controls, bounded evidence

The operations batch combines maintenance validation, retention, usage, migrations and release work. These controls add reviewable restriction and reporting surfaces, but their presence does not upgrade MemoryService into durable orchestration. The final schema includes sixteen migrations; the original test fixture applies only fourteen and misses agent_version introduced by migration sixteen. Separate full-schema teaching evidence repairs the fixture path without editing the historical application.

Historical excerpt: internal/maintenance/maintenance.go at 7985b9ebe276.

```
func Validate(window types.MaintenanceWindow) error {
        if window.Name == "" {
                return fmt.Errorf("name is required")
        }
        if _, err := cronjobs.Parse(window.Schedule); err != nil {
                return err
```

12. Release and site packaging: descriptions remain claims to check

The locked final documentation commit links installation, CLI, API, deployment and security guides from a reader-facing project page. Formatting fixes and release metadata are legitimate history entries, as are dependency merges and tracker edits. They are not noise to remove from the 105-commit count. Publication polish improves navigation; it cannot establish that the default executable runs HA, autoscaling or a durable database-backed control plane.

Historical excerpt: README.md at cbfc94ade5b5.

```
  <a href="https://alekpopovic.github.io/orch/#INSTALLATION.md">⬇️
Install</a> ·
  <a href="https://alekpopovic.github.io/orch/#CLI.md">🖥️ CLI</a> ·
  <a href="https://alekpopovic.github.io/orch/#GITHUB_PAGES.md">📚
Docs</a> ·
  <a href="https://alekpopovic.github.io/orch/#API.md">🔌 API</a> ·
  <a href="https://alekpopovic.github.io/orch/
#PRODUCTION_DEPLOYMENT.md">🚢 Deploy</a> ·
  <a href="https://alekpopovic.github.io/orch/#SECURITY.md">🛡️
Security</a> ·
```

Surviving tensions and unresolved evidence

The original instructions are ambitious about persistence, HA, autoscaling and production operations. The current call path provides a more precise classification: PostgreSQL and leader election are implemented packages, autoscaling is an unwired package, several networking/storage improvements remain design work, and the default server constructs MemoryService. These are not interchangeable degrees of production readiness. Inspect the final composition after studying an earlier diff so a historical feature addition does not become a present-day wiring claim.

Nine design-oriented prompt associations retain partial grades: 54, 56, 63, 68, 69, 72, 75, 77 and 79. Their missing evidence is explicit: a design file and related path changes do not demonstrate the entire requested architecture running. Execution transcripts establishing exactly which prompt was sent, in which environment and with which results are absent for the wider historical mapping as well. The atlas preserves that causality uncertainty even when a mechanism clearly exists today.

Developers should start with the domain, scheduler and reconciler checkpoints, then compare interface seams and the tests that constrain updates. DevOps engineers should follow agents, rollout and operational hardening, asking which observed states survive a process boundary and which cleanup actions reach a worker. Architects should compare PostgreSQL, shared late batches and final composition, asking where an intended consistency property has an actual enforcing caller.

A useful diff study ends with a falsifiable question: which new path changed behavior, which test exercises it, and what input or failure still falls outside that scope? That method preserves both the value of the project's history and the limits of what history alone can establish.

Appendix D – Worked Exercises and Design Solutions

A useful exercise produces an artifact another reader can inspect: a state trace, a calculation, a failing boundary, a test result or a recovery decision. It does not reward repeating the word reliable. The chapter exercises introduce mechanisms where they are taught; this appendix adds twenty-four substantial practice problems that combine those mechanisms and keep questions separate from worked answers.

There are eight developer, eight DevOps and eight architect exercises. Each has a stable ID, prerequisites, an estimated duration, a pinned source mechanism, a required artifact, hints and three objective scoring criteria worth two points each. Full credit is six points. A correct answer can conclude that the requested guarantee is absent; inventing missing persistence or enforcement loses credit even if the proposed design sounds plausible.

The eight developer entries are executable offline through existing pinned tests. Their tasks are source analysis and test-oracle interpretation, so no incomplete coding scaffold is required. Tests run in a separate module copy because Go internal-package rules and immutable evidence prohibit treating the Jekyll root as the application module. The GitOps test creates only a temporary local Git repository. None of these eight entries requires a Docker daemon, PostgreSQL connection or live registry.

Additional practice index

ID	Audience	Chapter	Required artifact
DEV-01	developer	10	A three-step claim trace and the isolated test log.
DEV-02	developer	14	A status/identity table plus the named local test log.

ID	Audience	Chapter	Required artifact
DEV-03	developer	18	Two guard evaluations and a cleanup trace plus the test log.
DEV-04	developer	17	A version/status budget table and the rollout test log.
DEV-05	developer	27	A two-namespace lookup matrix and the namespace test log.
DEV-06	developer	28	A race timeline and the quota test log.
DEV-07	developer	30	A revision/service-version trace and the local Git fixture test log.
DEV-08	developer	31	An occurrence/active-job table and both named test results.
OPS-01	devops	24	A recovery checklist with key/ciphertext ownership and stop conditions.
OPS-02	devops	3	A before/after identity inventory and re-enrollment decision record.
OPS-03	devops	39	A fully specified measurement sheet with run status and a claim boundary.
OPS-04	devops	05	A field-flow diagram from service through assignment to ContainerSpec and engine inspection.
OPS-05	devops	23	An authentication timeline with stored-hash/client-token divergence and safe stop condition.
OPS-06	devops	19	Two task/condition traces and a fencing risk decision.
OPS-07	devops	37	A three-layer health table with a verification probe for each layer.
OPS-08	devops	35	A canary/digest recovery plan with key backup, schema, downtime and rollback criteria.
ARC-01	architect	17	A budget calculation and counterexample timeline.

ID	Audience	Chapter	Required artifact
ARC-02	architect	5	A unit-correct table and an enforcement boundary statement.
ARC-03	architect	34	A labeled future-extension dependency graph and crash-test plan.
ARC-04	architect	34	A two-pool lock ownership diagram and a corrected release contract.
ARC-05	architect	26	A resolver/redirect policy diagram with a zero-hit loopback oracle.
ARC-06	architect	27	An authorization matrix for known/unknown/foreign IDs and a test-gap entry.
ARC-07	architect	25	A syntax/resolution/content verification ladder.
ARC-08	architect	38	A measurement contract with start/stop events, clock origin, censoring and scope.

Read the separate practice questions first, then compare your artifact with the worked practice solutions. A hint should help locate an enforcing boundary without handing you a fictional result. Estimated times assume the stated prerequisites are available; missing tools are a setup issue to record, not a reason to assert a pass.

How to attempt an offline exercise

Make a disposable copy of the locked Orch module and use Go 1.25.0 with its declared dependencies. Run the exact test filter named in the question. Preserve command, working directory, source SHA, Go version, UTC timestamps and exit code. Read the assertion that would reject the specified incorrect behavior; a package-level pass with zero matching tests is not completion.

The delivered exercise check actually ran all eight selected command groups and observed nine requested test passes, because the cron entry intentionally includes two tests. This is evidence for those deterministic

package fixtures. It does not establish Docker execution, cross-process durability or an actual production incident. A separate artifact grader accepts the serialized expected outcomes and rejects an intentionally wrong field for each executable entry; this checks grading discrimination. Separately, eight controlled semantic mutations in disposable source copies were each rejected by the selected package tests, as recorded in the mutation results. Those counterexamples demonstrate sensitivity to eight specified faults; they do not establish exhaustive mutation coverage.

When a fixture uses an injected clock, name that seam. The cron package does; not every subsystem exposes the same interface. Concurrent tests may guarantee one winner while leaving the winner's identity nondeterministic. Preserve that distinction in your answer instead of relying on timing or sleeping until a preferred goroutine wins.

Traces, calculations and operational artifacts

For a task trace, include ID, DesiredStatus, ActualStatus, owning NodeID, service/deployment version and relevant conditions. Conditions such as node_lost remain separate metadata. Show the initial state, message or policy decision, authorization check and resulting state. Do not move failed directly to healthy to simplify a diagram: replacement gets a new identity, while cleanup has its own permitted edge.

For resource arithmetic, keep CPU in integer millicores and memory in bytes. Five replicas at 250 millicores and 128 Mi reserve 1250 millicores and 671088640 bytes. State whether other namespace reservations exist before asserting a quota decision. Requested capacity, limits forwarded to Docker and observed process consumption are three different artifacts. Arithmetic can be correct while propagation is missing.

Operational answers need a bounded target, a verification step and a stop condition. A restore plan must retain the matching envelope key and compare canary content, not merely show a successful database

process exit. A memory-server restart plan must acknowledge that live intent was never written into that database. A node-loss plan must address an old process that can still run after its heartbeat expires.

Architecture answers should label proposals. A dependency graph connecting a durable service/store, transactional claim path and leader-loss cancellation is a future composition unless the source actually constructs it. Describe the smallest change and a falsifying crash window. A consensus paper or package interface supplies useful vocabulary; neither supplies proof that the default binary has activated HA.

Chapter exercise and answer map

Use this map to revisit each chapter's original exercises and independently worked companion. Chapter answers are separate from the twenty-four additional practice solutions, so readers can attempt either set without immediately exposing the other.

Chapter	Exercise context	Worked answers
1	Why Build a Container Orchestrator?	Chapter 1 answers
2	A Repository Built for Reasoning	Chapter 2 answers
3	Boot the Lab and Deploy the First Service	Chapter 3 answers
4	Domain Models and State Machines	Chapter 4 answers
5	Service Specifications, Resources, and Ports	Chapter 5 answers
6	Follow One Request Through the Control Plane	Chapter 6 answers
7	HTTP Contracts and Compatible APIs	Chapter 7 answers
8	A CLI That Operators Can Automate	Chapter 8 answers
9	Persistence, Transactions, and Optimistic Concurrency	Chapter 9 answers
10	A Deterministic Scheduler	Chapter 10 answers
11	Reconciliation as Repeated Convergence	Chapter 11 answers

Chapter	Exercise context	Worked answers
12	Idempotent Docker Operations and Drift	Chapter 12 answers
13	Agents, Heartbeats, and Task Reports	Chapter 13 answers
14	Health Checks That Respect Trust Boundaries	Chapter 14 answers
15	Events, Audits, and Notification Boundaries	Chapter 15 answers
16	Log Streaming Without Losing Control	Chapter 16 answers
17	Rolling Updates and Rollback Invariants	Chapter 17 answers
18	Safe Deletion and Resource Cleanup	Chapter 18 answers
19	Offline Nodes, Draining, and Rescheduling	Chapter 19 answers
20	Endpoint Discovery and Internal DNS	Chapter 20 answers
21	Ingress, Host Networking, and the Overlay Roadmap	Chapter 21 answers
22	Autoscaling Decisions and Missing Integration	Chapter 22 answers
23	Authentication, Authorization, and Node Trust	Chapter 23 answers
24	Encrypted Secrets and Configuration Redaction	Chapter 24 answers
25	Registry Credentials and Immutable Image Identity	Chapter 25 answers
26	Security Contexts, Admission, and Policy	Chapter 26 answers

Chapter	Exercise context	Worked answers
27	Namespaces and the Limits of Multi-Tenancy	Chapter 27 answers
28	Quotas and Usage Accounting	Chapter 28 answers
29	Supply Chain Designs Without False Guarantees	Chapter 29 answers
30	GitOps, Drift, and Controlled Reconciliation	Chapter 30 answers
31	Finite Jobs and Time-Based Work	Chapter 31 answers
32	Local Volumes, Claims, and Workload Topology	Chapter 32 answers
33	Extension Interfaces and a Plugin Roadmap	Chapter 33 answers
34	Leader Election and the Durable HA Boundary	Chapter 34 answers
35	Upgrades, Migrations, Maintenance, and Retention	Chapter 35 answers
36	Fakes, Property Tests, Fuzzing, and Contracts	Chapter 36 answers
37	Failure Injection and Recovery Evidence	Chapter 37 answers
38	Metrics, Dashboards, and Actionable Signals	Chapter 38 answers
39	Performance and Scale Tests That Say What They Measure	Chapter 39 answers
40	Release a System You Can Honestly Operate	Chapter 40 answers

Self-review and instructor guidance

First attempt a question using only its stated prerequisites. If the solution depends on an unstated constructor, clock, namespace context or resource unit, revise the question before grading. The second pass should compare every rubric criterion against the actual artifact, not against an impression that the answer sounds experienced. For a trace, distinguish wrong ownership from wrong arithmetic: both matter, but they require different feedback.

An operational artifact earns credit when another person can identify the exact disposable target, evidence origin, decision point and safe stop. It does not need to mutate a real service. An architecture artifact earns credit when it specifies state owners, failure domains and a test that could disprove the desired property. A proposal that simply adds PostgreSQL or a leader flag without changing callers is incomplete.

These exercises can be adapted to a future durable extension by adding explicit store construction, schema/key recovery and leadership fencing assumptions. Retain the original source-derived answer beside the extension answer. That comparison teaches how composition changes a guarantee without rewriting history or silently repairing the immutable baseline.

The exercise catalog is versioned with stable IDs and chapter links. Keep answers explicit about inspected, executed, expected and proposed evidence. Future revisions may change code and prerequisites; they must not turn the current edition's bounded observations into universal production promises.

Source-backed boundary statements for this appendix are indexed in the appendix claim record. Detailed command, state, exercise and reference tables retain their own source provenance.

Appendix E – Operator Runbooks and Measurement Contracts

An operational procedure needs a supported mode, an observation and a stop condition. Otherwise a useful diagnostic command can become an uncontrolled repair attempt. These runbooks target disposable Atlas laboratories and the pinned implementation. The default server keeps live state in memory; the database restore procedure applies to an explicitly constructed extension, not to that default process.

The companion index contains ten complete playbooks with trigger, real command paths, triage order, decision points, bounded action, verification and rollback or stop condition. The condensed table is a navigation aid. Supply the dedicated namespace and captured resource IDs from the laboratory rather than pasting placeholders into a shared cluster.

Incident	First observations	Decision that prevents a misleading repair
pending-tasks	`orch service inspect`, `orch service ps`	If no eligible capacity exists, distinguish a valid pending result from a failed scheduler loop; ordinary immediate placement can bypass Plan constraints.
unhealthy-workload	`orch service ps`, `orch service inspect`	Decide whether unhealthy stays retryable or MemoryService converts it to failed under replacement policy; new rollout success requires healthy.
agent-enrollment	`orch node ls`, `orch node inspect`	Distinguish incorrect authority, rotated-token response loss, wrong-node access and a memory-server restart that erased credential records.
node-loss	`orch node inspect`, `orch service ps`	Stateless replacement is eligible; stateful work remains local for manual recovery. Neither implies the old process stopped.

Incident	First observations	Decision that prevents a misleading repair
drain-capacity	`orch node drain-status`, `orch node inspect`	If replacement capacity is unavailable, waiting is correct; drain complete concerns desired work and can precede physical removal.
rollout-failure	`orch rollout status`, `orch service inspect`	Choose between waiting for an owned probe repair and an explicit rollback; a raw running target is not newHealthy.
gitops-drift	`orch gitops ls`, `orch gitops status`	A fetch/parser failure should not authorize pruning untrusted broad paths; distinguish drift observation from a successful apply.
quota-denial	`orch quota get`, `orch service ls`	Zero quota fields mean unlimited; a racing mutation can legitimately win before this request. Requested reservations are not measured Docker consumption.
memory-restart	`orch service ls`, `orch node ls`	This is expected process-local loss, not proof PostgreSQL restore failed; surviving containers need careful label reconciliation.
postgres-restore	`orch cluster check-upgrade`	A successful pg_restore exit is insufficient; the current memory binary does not read this database as live intent.

Process readiness and control-loop evidence

`GET /healthz` and `GET /readyz` are process responses. They do not test durable storage, validate every worker's reachability or prove that desired replicas are serving users. The optional schema check can fail startup on an incompatible database; it does not turn readiness into continuous database health or replace the live MemoryService constructor.

The executable starts GitOps, cron, scheduler, rollout and node monitoring, plus usage capture and retention work. DNS is conditional on its configured listener; profiling uses a separate explicitly enabled listener. It does not start the standalone reconciler, autoscaler or leadership package. Their registered collectors describe available instrumentation contracts, not proof those loops are active. A zero or absent sample must be interpreted according to that wiring.

Use counters for accumulated occurrences and gauges for current values. API request counters and histograms have method, route and status labels. Agent Docker operation counters use operation labels; task-state change counters use status. The healthcheck success/failure counters have no label keys. Do not invent a namespace label because a dashboard would find it convenient. The companion collector inventory records exact names, types and label keys from source.

An alert rule or suggested threshold remains a proposal until its collector, scrape path, scope and action were configured and tested. A service-level objective additionally needs a user-facing success definition and an observation window. A scheduler counter cannot substitute for gateway request availability, and requested CPU cannot substitute for Docker consumption.

Measurement contracts

Question	Unit	Start/denominator and finish/numerator
placement-success	ratio	Cohort snapshot to first assigned observation
task-convergence	seconds	Accepted service version to first complete matching healthy population
replacement-latency	seconds	Server node_lost/failed decision to replacement actual healthy
rollout-health-budget	tasks	Every controller snapshot until terminal deployment
heartbeat-age	seconds	Heartbeat update to each monitor observation
drift-recovery	seconds	Observed drift state to successful matching apply/version
quota-usage	millicores and bytes	Same namespace mutation/snapshot boundary
cron-occurrence-uniqueness	count per scheduled instant	Occurrence selection to persisted LastScheduleAt/created job identity

Question	Unit	Start/denominator and finish/numerator
backup-recovery-completeness	ratio and seconds	Confirmed canary backup boundary to successful restore+verification

Each contract explicitly records clock origin, sampling, workload size, warm-up, missing-data handling and limitations. Local elapsed durations should use one monotonic observer clock where possible; preserve server UTC timestamps for ordering and interpretation. Never subtract monotonic readings from separate processes. When wall-clock comparisons are unavoidable, record synchronization assumptions and clock uncertainty.

Missing completion is censored or timed out, not a zero latency sample. A denominator must identify the same cohort as its numerator. For quota usage, zero means unlimited rather than a division operation. For cron, occurrence identity includes the cron resource and scheduled instant; it does not prove an external side effect ran exactly once. For backup recovery, matching count alone is insufficient: compare stable IDs, content digests, schema and synthetic secret decryptability.

A bounded, actual fake-agent experiment

The following command was executed from the separate disposable Orch module using the source-built Go 1.25.0 binary. It performs **in-memory fake-node/status simulation**. It does not pull an image, start Docker containers, use PostgreSQL or measure network serving throughput.

```
orch-loadtest --nodes 3 --services 12 --replicas 2 \
  --failure-rate 0.1 --duration 1s --seed 17
```

There was no separate warm-up: initial service creation contributes to the reported convergence timing, then a one-second simulation follows. The recorded host was Linux x86_64 with 12 logical CPUs; the evidence file retains the exact kernel/glibc platform and Go version. That machine description is context, not a portable capacity estimate.

Actual output was 24 tasks created, 8 running and 16 failed at the final snapshot, 10 simulation iterations, 12 initial reconciliation iterations and zero reported scheduler/reconciler errors. The timing fields are retained in raw evidence; their microsecond scale must not be presented as Docker startup latency. A deterministic random seed controls failures, but scheduler timing and wall-clock measurement remain machine dependent. A zero exit means this bounded simulation ran, not that all tasks converged healthy.

A future durable-cluster plan must first name an explicitly wired service/store, transaction and leadership paths, exact schema and credential recovery. Then measure accepted intent through real worker execution separately from database throughput and backup reconstruction. Use captured canaries and failure windows; do not attach performance numbers from this fake run to that future architecture.

Recovery and laboratory navigation

The restore playbook uses a new disposable target and all sixteen migrations. Preserve the envelope key independently: restored encrypted rows cannot be decrypted from KeyID alone. Stop on schema or canary mismatch and retain the original target. Database success cannot recover MemoryService maps that were never written there, and database backups do not contain Docker volumes or container filesystems.

Each laboratory records its own actual or expected evidence class. Follow the eight laboratory guides for initialization, placement, drift, rollout, security, namespaces, GitOps/cron and recovery. They should be read alongside the corresponding playbook; a source-derived procedure is not a transcript that the action was already performed on your machine.

- Initialization and restart
- Scheduler placement
- Reconciler drift
- Rollout and rollback
- Security boundaries
- Namespaces and quotas

- GitOps, jobs and cron
- Recovery boundaries

The boundary map is deliberate: pending/drain incidents test placement; unhealthy work and rollout test runtime execution; enrollment and node loss test identity; GitOps and quota test intent mutation; memory restart tests process-local ownership; restore tests the database extension. Crossing one boundary successfully does not close every other incident path.

Sources: operations, registered collectors, server composition, fake-load implementation, and backup/restore scope.

Source-backed boundary statements for this appendix are indexed in the appendix claim record. Detailed command, state, exercise and reference tables retain their own source provenance.

Appendix F – Glossary, Bibliography, and Boundary Index

Terminology is useful when it keeps neighboring guarantees separate. Process readiness differs from workload health; requested resources differ from observed use; local durability differs from replicated availability. This glossary gives each term a conceptual meaning and a pinned Orch interpretation. A future-design label is not a promise that the current binary implements the concept.

The complete reference includes stable IDs, related terms, numbered chapter slugs and immutable source provenance. Use the source link to inspect a mechanism, then the chapter to understand its failure window. Definitions avoid circular explanations by naming the actual owner, field or consumer. They describe this edition rather than a changing default branch.

Mechanism glossary

Term	Meaning and Orch boundary	Chapter
Trust boundary	A crossing between differently authorized actors or data. User JWT, node credentials, authorized task secrets and Docker authority require separate checks.	23
Desired state	The requested target of control. Server-owned Task.DesiredStatus constrains which worker reports remain acceptable.	4
Actual state	The latest accepted observation of execution. Assigned agents report task progress; policy and node-loss logic can transform stored results.	13
Reconciliation	Repeated comparison and action toward intent. The standalone reconciler exists, while MemoryService also owns integrated bookkeeping paths.	11

Term	Meaning and Orch boundary	Chapter
Convergence	An observed approach toward a stated target. Capacity, policy and successful worker actions are prerequisites, not guaranteed eventual outcomes.	11
Idempotency	Retrying an operation without duplicating its intended effect. Repeated task/container identity checks and duplicate-report rules provide scoped mechanisms.	12
Optimistic concurrency	Rejecting a write when its expected version is stale. Store operations compare expected UpdatedAt rather than trusting a prior read.	9
Task claim	A conditional assignment of pending work. One scheduler/store write selects NodeID; it does not fence an isolated old process.	10
Advisory lock	An application-coordinated database lock. The leadership package uses session-owned PostgreSQL locks; original-connection release matters.	34
Split brain	Concurrent actors believing they may own the same responsibility. Heartbeat suspicion and package leadership do not establish enforced single-writer runtime authority.	34
Terminal state	A state that cannot resume ordinary execution. Failed/stopped tasks allow cleanup to removed; terminal deployments remain immutable histories.	4
Namespace	A named scope for resource selection and access. Context-scoped service lookups do not prove network isolation or every newer workload path.	27
Admission	Validation and policy before accepting intent. ClusterPolicy can restrict workload fields; acceptance still differs from runtime field propagation.	26
Quota	A bound on scoped requested resources or objects. MemoryService serializes namespace accounting; zero limits mean unlimited.	28
Credential enrollment	Exchanging initial authority for an identity credential. Bootstrap registration issues a hashed-record node token for subsequent authenticated calls.	23

Term	Meaning and Orch boundary	Chapter
Secret envelope	Authenticated encryption wrapped with key metadata. LocalEnvelope uses AES-GCM; KeyID does not replace the required decrypting key.	24
SSRF	Server-side requests redirected toward unintended destinations. Probe targets and redirect/dial behavior need policy checks beyond a friendly URL string.	14
Local volume	Persistent data attached to one runtime locality. Node-pinned volume metadata does not replicate data or establish safe stateful failover.	32
Rollout surge	Extra active work permitted during a version transition. MaxSurge bounds active task accounting above desired replicas, not physical cleanup completion.	17
Process readiness	A process-reported ability to accept its expected traffic. The readyz handler does not validate durable orchestration or healthy workload replicas.	7
Workload health	A workload-specific observation of acceptable behavior. Probe thresholds affect healthy/unhealthy states; gateway reachability remains a separate observation.	14
Resource request	Declared capacity used to plan or admit work. CPU uses integer millicores and memory uses bytes after parser conversion.	5
Resource limit	A runtime bound intended to constrain consumption. The Docker adapter supports limits, but the ordinary agent path omits their propagation.	5
Resource reservation	Accounted requested capacity rather than measured use. Quota and placement arithmetic must not be described as Docker usage or billing.	28
Memory server	A control plane whose live maps belong to one process. The default server constructs MemoryService and loses intent/credentials on restart.	1
Durability	Retention of committed state across a defined failure. PostgreSQL store support is not default live-state wiring; identify the actual owner.	9

Term	Meaning and Orch boundary	Chapter
Replicated availability	Continued service through failures using multiple coordinated copies. Local persistence and an unwired leader package do not supply this system property.	34
Leader lease	A bounded authority claim with a loss signal. Lease.Done closing does not alone prove the original database session released its lock.	34
Fencing	Preventing stale owners from performing side effects. Physical worker/storage fencing is future work, not an effect of marking a node offline.	34
Autoscaling	Changing replicas in response to a policy and measurement. The package computes recommendations but is not started by the default server.	22
Stabilization window	A policy interval intended to damp rapid scaling decisions. The field is accepted and validated, but its declared window is not consumed by decisions.	22
Metrics provider	A component supplying a defined measurement to a policy. Autoscaler CPU input is an interface; a real provider is not wired into the binary.	22
Cooldown	A delay before another allowed scaling decision. The autoscaler remembers last scale time in a process-local map.	22
Parser resolver	A resolver that interprets an existing image reference. It accepts supplied digest syntax without fetching a registry's remote digest.	25
Digest pinning	Naming content with an immutable digest-form reference. Parser/admission support does not prove remote manifest retrieval or layer verification.	25
Supply chain	The path from source/artifact origin to executed content. Signing, SBOM and remote provenance designs require separate implementation and evidence.	29
Registry credential	Authority for access to an image registry. Encrypted stored credentials are resolved for authorized task execution; metadata excludes secrets.	25

Term	Meaning and Orch boundary	Chapter
Task identity	A stable ID for one execution attempt. Replacement creates another ID instead of resurrecting a failed attempt.	4
Deployment version	A service intent generation used during rollout. Tasks are compared against target versions; rollback creates another deployment history record.	17
Restart policy	A rule determining recovery after workload failure. MemoryService can convert unhealthy to failed when service replacement is permitted.	14
Probe	A bounded health observation against a configured target. HTTP/TCP policy, timeout and thresholds belong to the actual health implementation.	14
Public port	A worker-level port forwarding traffic toward a container. Task-assigned public ports can differ from service requests and agent advertise ports.	20
Node advertise address	The address a worker publishes for reaching its services. Traefik extracts its host and combines the task's published port.	21
Discovery	Returning endpoints describing presently eligible work. HTTP discovery filters desired-running tasks; its DNS path has a distinct filter.	20
DNS answer	A cached name-to-address response. The local DNS implementation returns deduplicated node IPv4 addresses without task ports.	20
Ingress	A gateway path from client routing to workload endpoints. Traefik uses node host/public-port HTTP backends; it does not create an overlay.	21
TLS boundary	The hop where encryption terminates or begins. A TLS router flag does not automatically encrypt the generated HTTP worker backend.	21
Stateful workload	Work whose data/identity constrains relocation. Offline stateful tasks retain assignment for manual recovery rather than assuming local data moved.	19

Term	Meaning and Orch boundary	Chapter
Task condition	Additive metadata describing a task circumstance. node_lost remains separate from actual and desired lifecycle states.	19
Drain	Moving desired work away from a selected worker. Completion counts desired work; actual owned-container removal can finish later.	19
Node offline	A control-plane status indicating loss of acceptable heartbeat presence. Expiry marks suspicion and recovery intent; it cannot stop a partitioned process.	19
Heartbeat age	Elapsed server time since a retained heartbeat observation. Zero LastHeartbeat is handled separately; wall-clock assumptions and cutoff strictness matter.	19
Cache TTL	A time interval for retaining a cached response. DNS uses a configured TTL, creating a freshness boundary after task changes.	20
Credential rotation	Replacing a credential while retaining an identity relationship. Lost heartbeat responses can leave the worker token behind the stored server hash.	23
Credential revocation	Rejecting an identity's previously issued credential. The stored flag participates in validation; new credential storage changes that record.	23
Bootstrap authority	Initial authority used to admit a worker. Registration and the agent log listener use it differently from issued control-plane credentials.	23
JWT	A signed token carrying claims validated by the receiver. User validation is conditional on signing-secret configuration; node tokens are a separate scheme.	23
RBAC	Authorization based on an assigned role. Viewer/operator/admin checks depend on configured user JWT; newer routes can fall through to admin.	23
Namespace role	Role authority scoped to a selected namespace. Static user maps replace global role and clear namespace-role claims in the current middleware.	27

Term	Meaning and Orch boundary	Chapter
Configuration precedence	An ordered selection among configuration inputs. Nonempty flags override environment, YAML and defaults through the corresponding loader.	24
Redaction	Removing sensitive content from an exposed representation. Config print replaces sensitive fields; it does not sanitize all logs or workload output.	24
Audit record	An account of an attempted operation and its outcome. Best-effort append behavior must not be mistaken for transactional external audit delivery.	15
Event	A structured observation of a control-plane occurrence. State-change events and duplicate-report rules differ from a durable event-stream guarantee.	15
Retention	A rule for pruning older retained history. MemoryServer history pruning is separate from a durable backup and restore strategy.	35
Usage snapshot	A sampled accounting record over defined resources/time. Requested task resource accounting is not measured consumption or a validated billing engine.	28
Log stream	A bounded or followed flow of task output. API/agent stream cancellation, authorization and container ownership remain distinct checks.	16
Cancellation	A signal that dependent work should stop. Context cancellation requires each loop/stream/lease consumer to release its own resources.	16
Lost response	An operation result that the requester never receives. A heartbeat credential can rotate before its caller observes the new token.	23
Cleanup	Removing runtime objects after execution intent ends. Label checks protect ownership; service deleted waits for actual removed states.	18
Duplicate report	A repeated observation of the same accepted state. Identity transitions are accepted without manufacturing another state-change event.	13

Term	Meaning and Orch boundary	Chapter
Atomicity	A defined set of effects that succeeds or fails together. Local mutex mutation and store transaction behavior have different process-failure boundaries.	9
Availability budget	The minimum healthy/available population a controller protects. Old running/healthy and newHealthy predicates differ during rollout.	17
Rollback	A new operation restoring an earlier service version. It preserves failed/succeeded terminal deployment records instead of rewriting history.	17
Runtime adapter	The interface boundary translating intent into engine calls. Docker support in isolation does not prove ordinary agent field propagation.	12
Fake runtime	A controlled test double for runtime side effects. Passing fake-agent tests does not measure Docker startup or registry behavior.	36
Measurement contract	A definition of cohort, events, clocks and missing data. Fake-load timing and real runtime convergence require separate measurement scopes.	39
Artifact backup	A retained copy of files or database records. A database dump excludes runtime volumes and the separate envelope key.	40
Key loss	Loss of the material required to decrypt retained ciphertext. Restored rows and KeyID alone cannot recover LocalEnvelope plaintext.	24
Recovery point objective	A chosen tolerance for lost committed changes. A future durable deployment must define it; the memory server supplies no durable checkpoint promise.	40
Recovery time objective	A chosen target duration for restoring a defined service. Measure verified recovery, including schema/key/canary checks, rather than database exit alone.	40
Teaching extension	Book-owned runnable work with separately stated changes. It must not silently repair or inherit unrun claims from the immutable baseline.	36

Term	Meaning and Orch boundary	Chapter
Source pin	An exact revision and byte inventory defining an edition. Full-SHA citations distinguish current baseline code from historical implementation snippets.	2
Claim classification	A label describing a capability's actual integration status. Runtime-integrated, package, partial, design, teaching and future labels prevent compositional overclaims.	1

Annotated primary bibliography

Repository citations are evidence about this edition. Vendor documentation supplies the external protocol/runtime contract, and research papers supply comparisons. A paper about consensus cannot establish consensus in Orch. Docker documentation cannot establish that the ordinary agent forwards resource limits. Follow the call path and scoped execution record before making either claim.

The locked module declares Go 1.25.0, Docker SDK v28.5.2, pgx v5.10.0, Prometheus client v1.24.1, Cobra v1.10.2 and yaml.v3 v3.0.1. These are dependency versions, not an inferred Docker daemon negotiation or PostgreSQL server version. All repository bibliography entries use full SHAs; moving external pages retain their access date instead of pretending to be immutable snapshots.

Source code/history

- Orch source edition, Aleksandar Popovic, revision `cbfc94ade5b57e449d6b73bf1485b1c5b1647022`. Read executable composition before interpreting feature directories. The companion bibliography indexes every individual cited file and historical commit.

Go/Docker contracts

- The Go Memory Model, The Go Authors. 2022-06-06 specification revision. Accessed 2026-10-13. Synchronization vocabulary and local memory visibility; does not establish distributed durability.
- context package, The Go Authors. Go 1.25.0. Accessed 2026-10-13. Cancellation propagation and ownership of cleanup; a canceled context alone does not release every external resource.

Distributed systems

- In Search of an Understandable Consensus Algorithm (Extended Version), Diego Ongaro and John Ousterhout. Extended paper; retrieved PDF edition. Accessed 2026-10-13. Useful contrast for leader election and replicated-log assumptions; Orch does not implement Raft by citing this paper.

HTTP/security

- Docker Engine security, Docker Inc. Unversioned official documentation. Accessed 2026-10-13. Explains daemon/socket trust and container security boundaries; does not prove Orch passes each runtime field.
- RFC 9110: HTTP Semantics, R. Fielding, M. Nottingham, J. Reschke (editors). RFC 9110 / STD 97, June 2022. Accessed 2026-10-13. Defines protocol semantics; application idempotency and asynchronous deletion still require their own contracts.
- Server Side Request Forgery Prevention Cheat Sheet, OWASP Cheat Sheet Series. Unversioned official guidance. Accessed 2026-10-13. Frames destination-policy and request trust boundaries; actual probe behavior remains verified against pinned source/tests.

PostgreSQL/concurrency

- 13.3. Explicit Locking, PostgreSQL Global Development Group. PostgreSQL 16. Accessed 2026-10-13. Distinguishes row, transaction and session-level advisory lock ownership used in the leadership audit.

Observability

- Instrumentation, Prometheus Authors. Unversioned official guidance. Accessed 2026-10-13. Explains instrument selection and scoped metrics; a registered collector does not establish an active controller.

Operating practice

- Backup And Restore, Orch repository edition. Use its key/database artifact distinctions while preserving the default memory-server boundary.

Reading paths through the references

Developers can study six entries in this order: the source composition to locate owners; lifecycle helpers to find valid edges; scheduler tests to identify a race oracle; Go memory model to separate synchronization from distributed persistence; context to follow cancellation; and HTTP semantics to distinguish protocol retries from application effects.

DevOps engineers can follow server composition for active loops; agent runtime construction for engine authority; Docker security for daemon trust; Prometheus instrumentation for observation scope; backup/restore for excluded artifacts and keys; and PostgreSQL locking for the lifetime of database ownership.

Architects can compare architecture intent with actual composition, then the historical graph, leader implementation, Raft's extended paper, and OWASP's SSRF guidance. The questions are which owner is real, which failure domain is covered, and which extra mechanism an improvement still requires.

Version and evidence method

A source-inspected claim identifies code and tests but does not claim a run. Executed evidence additionally records commands, prerequisites, versions, UTC timestamps, exit status and logs. Historical snippets use their own commit/object identities; teaching extensions identify changed behavior and independent checks. Expected outcomes remain predictions until the matching experiment runs.

The bibliography checker resolves every manuscript URL to an entry, validates repository files/commits against the locked inventory or separately verified historical objects, and records external access outcomes. A link that opens is not proof that a technical claim is correct. Revisit the enforcing symbol and the integration boundary when a future Orch edition changes, rather than carrying this glossary's implementation interpretation forward unchanged.

Source-backed boundary statements for this appendix are indexed in the appendix claim record. Detailed command, state, exercise and reference tables retain their own source provenance.

Lab 01 — Initialize, deploy, and distinguish readiness from durability

This lab asks a narrow operational question: does starting a fresh default `orch-server` restore the objects that a previous server owned? A PostgreSQL connection string is configuration; the relevant implementation is the constructor that chooses live state. We inspect that constructor, exercise a complete disposable workload, and then measure a fresh process independently. These experiments use the frozen source edition, not the website's Jekyll root as a Go module.

There are three useful modes. **Inspection** needs only Python and the installed source corpus. **Offline execution** needs the exact Go toolchain, previously downloaded dependencies and permission for temporary loopback test listeners. **Docker execution** additionally needs a lab-authorized Docker daemon and the cached `nginx:1.27-alpine` image. PostgreSQL migration observations are a fourth, separate track. A skip in one mode never becomes a pass in another. The commands in this guide are procedures; the linked records identify what the author actually ran.

Establish the source and ownership boundary

From the website root, run `python3 books/inside-orch/scripts/verify_pack.py`. The source lock names full revision `cbfc94ade5b57e449d6b73bf1485b1c5b1647022` and 361 application files. Copy that source to a new disposable directory outside `books/inside-orch`. Never build inside `evidence/orch`. Inspect `cmd/orch-server/main.go` at the locked revision: it constructs `controlplane.NewMemoryService` regardless of `DATABASE_URL`. The optional schema check opens the migration manager but does not replace that constructor.

Choose loopback API addresses dynamically, record every child PID, and stop only those children. The supplied `scripts/observe_process_restart.py` owns two sequential server processes on

an allocated loopback port, allowlists their environment, checks the source-copy hashes, bounds readiness polling, and records stop results. It does not connect to an existing server. If the allocated port becomes occupied before startup, the child exits and the experiment fails; it must not attach to whatever owns that port.

The source export uses Go1.25.0. Set `GOROOT` to the selected toolchain's root, `GOTOOLCHAIN=local`, `GOENV=off`, and run-owned `GOPATH`, `GOMODCACHE`, and `GOCACHE`. For a source export without `.git`, set `GOFLAGS=-buildvcs=false`. Build `cmd/orch`, `cmd/orch-server`, and `cmd/orch-agent` into the disposable run's `bin` directory. Record those commands and tool versions rather than assuming a globally installed `orch` matches the source lock.

Verify the actual command and manifest surfaces

Run the built CLI with `version` and `--help`, then the built server with `config print`. The version record must match the inspected CLI; help is discovery, not evidence that every listed command ran. Configuration output must redact the database URL, bootstrap token and envelope key. An empty JWT setting means optional user authentication is disabled in this local fixture. Do not transplant that setting into a public listener.

Validate book-http.yaml using `orch validate <absolute-file-path>`. This is the CLI's deployment schema: `resources` and `restart` are supported keys. The HTTP API instead accepts a `spec` wrapper with domain JSON field names. They are distinct surfaces. The first authoring attempt used API names in CLI YAML and was rejected; the corrected manifest passed the real parser. Preserving this failure makes the difference visible.

The CLI fixture requests 100m CPU and 128Mi memory, with one replica and no published ports. The separately instrumented HTTP resource experiment deliberately uses 100m/64Mi requests and 250m/128Mi limits to distinguish request accounting from Docker enforcement. Do not combine those two inputs into a single claimed transcript.

Observe a complete, bounded Docker flow

The Docker teaching harness starts a fresh loopback API handler and a real agent runner with the locked adapter. Its wrapper permits cached images only and restricts stop/remove to captured container IDs. It adds a node filter to task-only log lookups for this disposable exercise. Those safety instrumentation changes are teaching code; they do not assert the upstream runtime already has the same restrictions.

The harness records node, service, task and container identifiers from actual responses. It checks the nginx welcome body with `docker exec ... wget http://127.0.0.1/` inside the captured container, reads logs through server→agent→Docker, verifies one service-created event, scales 1→2→1, waits for the removed task, deletes the service, and waits for terminal removal of both historical tasks. No workload port is published on the host. Its timelines contain observed task objects and UTC times, not invented identifiers. The API uses `httptest` here; this is a package integration flow, not execution of upstream Compose or proof that the production server's controllers are durable.

The actual successful run is docker-complete-flow-log-guard.json, with raw structured output in its stdout. An earlier wrapper rejected the log handler's task-only lookup; that unsuccessful harness attempt remains recorded separately. Do not attribute a harness safety-filter mistake to the source application.

Restart a real default server process

Run `scripts/observe_process_restart.py` with explicit `--working-copy`, `--server`, and a new book-relative `--output`. It captures `/v1/version`, creates `book-payments`, registers a manual lab node and creates `book-http`. It records services, tasks, nodes and namespaces before stopping the first child. Manual registration does not execute a Docker workload. The second child is a new binary process with the same safe configuration and a new MemoryService.

The actual restart record shows readiness before and after, a service/task/node present before restart, empty lists afterwards, disappearance of the custom namespace and recreation of `default`. The captured PIDs differ. The experiment intentionally leaves agent and Docker survival out of scope; the previous Docker flow already cleaned its own containers. This isolates process-owned intent from remote side-effect survival. A separate combined restart experiment keeps one explicitly owned Docker container running across two actual server processes. The first server receives a manually authenticated running-task report with that captured container ID. The fresh server loses the service, task and node while the exact container remains running; cleanup then confirms its absence. This is an observed orphan boundary, not agent Runner adoption or restored intent.

Migration status is measured independently in full-schema recovery evidence: all 16 migrations were applied in an owned throwaway database. The restart fixture points to an unused local database endpoint with schema checking disabled. These records jointly establish two separate mechanisms; they do not claim that the restart process connected to the migrated database or restored its rows.

Verification and cleanup

Run `python3 books/inside-orch/scripts/verify_lab_records.py --lab 01 --mode inspection` for a useful source-only conclusion without Go downloads or Docker. Use `--mode recorded` to check the author-run evidence and write `validation/labs/01-check.json`. Recorded mode audits existing results; it does not silently rerun containers or overwrite a reader's experiment. A new execution should use new output filenames and explicitly select its own target environment.

Stop the captured agent before deleting its containers. The post-run inspection confirms all three captured Docker IDs are absent. The harness removes only its captured IDs; never prune the Docker daemon, kill all processes with a matching name, or delete every `orch.managed=true` container. A server process exit is not permission to

remove containers from another run. For an interrupted run, re-establish PID command lines and container identity before cleanup. Preserve observations and failure logs before removing temporary files.

Questions with worked answers

1. **Why can `/readyz` pass after all services disappear?** Readiness describes the new server's ability to handle its configured work. It does not assert that a previous process's desired-state maps have been reconstructed. The before/after object lists are the durability oracle.
2. **Would successful `pg_restore` change this result?** It can restore the PostgreSQL store's rows, schema and encrypted values. The default constructor still selects MemoryService, so restoration alone does not hydrate live orchestration intent. A durable-service composition would be a separately implemented and tested extension.
3. **What if an agent and container survive the API restart?** A remote container can outlive the memory that authorized it. Re-registration, ownership matching and cleanup need their own experiment, including stale credentials and namespace identity. The isolated restart record cannot establish that result. The combined experiment observes survival of a manually associated container, while actual Runner re-enrollment and adoption still require separate evidence. Broad cleanup would destroy the evidence needed to diagnose the boundary.

Lab 02 - Deterministic placement and competing claims

Atlas now has six pending candidates and three advertised nodes. Before running anything, decide which candidates fit, which node wins, and which tasks should remain pending. The purpose is to distinguish three contracts: a deterministic plan, one accepted claim for a task, and the runtime limits on a real container. No one result establishes all three.

Environment and ownership

Use the source edition `cbfc94ade5b57e449d6b73bf1485b1c5b1647022`, Go 1.25.0 with its matching GOROOT, and a disposable Orch copy. The teaching program imports actual internal packages and therefore runs beneath that module. It starts no listeners, containers or database connections. Its IDs and timestamp are fixed model inputs; they are not identities captured from a live cluster.

The complete input is fixtures/plan.json. It uses Go's `scheduler.PlanInput` representation, not a deployment YAML or public HTTP body. The separate teaching program reads that file and rejects the wrong source revision or candidate count. Copy the program unchanged into `.book-labs/lab02/main.go` in the disposable source tree. Record the source and fixture hashes before execution. Do not build inside `evidence/orch` or modify application files to change the outcome.

Predict the plan

CPU values below are millicores; memory values are binary mebibytes. The planner uses allocatable resources and requests. Capacity is deliberately larger than allocatable, and limits are twice requests, so confusing those fields changes the calculation.

Node	Status and label	Capacity	Allocatable	Existing work
a	offline, zone=green	8000m / 8192Mi	4000m / 4096Mi	None
b	ready, zone=blue	2000m / 1024Mi	1000m / 512Mi	100m / 64Mi, TCP31080
c	ready, zone=blue	2000m / 1024Mi	1000m / 512Mi	None

The existing task on b has desired running and actual stopping. That combination is intentionally still active according to the shared helper. It reserves its request and port in this model even though its process is progressing toward stop. This is not a claim that every desired-stop state remains accounted until Docker confirms physical termination; the distinction is tested below.

Candidate	Request	Required zone	Published TCP port
atlas-01	400m / 128Mi	blue	31080
atlas-02	400m / 128Mi	blue	Dynamic
atlas-03	400m / 128Mi	blue	Dynamic
atlas-04	100m / 64Mi	green	Dynamic
atlas-05	2000m / 1024Mi	blue	Dynamic
atlas-06	400m / 128Mi	blue	31080

Write the selected node and the reason for every rejection before reading the observed result. Plan processes candidates in stable task-ID order. Eligibility precedes scoring: ready status, labels, requests, mounts and ports can reject a node regardless of its attractive memory score. Among eligible nodes, more free memory after placement wins, followed by fewer same-service tasks, fewer total tasks and lexical node ID.

The predicted first choice is c for atlas-01. Node a is offline and b's fixed port is occupied. After this choice c has 384Mi free, while b still has 448Mi. Atlas-02 therefore selects b and receives dynamic TCP30000.

Now b has 320Mi free, so atlas-03 selects c and receives TCP30000 on that different host. Reusing a numeric published port across hosts is valid in the planner's node-scoped model.

Atlas-04 has no ready green node. Atlas-05 exceeds both ready nodes' allocatable resources. Atlas-06 cannot acquire TCP31080 on either ready node. All three remain pending. A correct no-fit result is useful feedback, rather than an exception to suppress or an invitation to ignore constraints.

Execute and inspect the model

From the disposable source root, set `BOOK_ROOT` to your canonical book directory and use the pinned Go executable. The variable is a reader-selected path, not a special application setting.

```
go run ./.book-labs/lab02 "$BOOK_ROOT/labs/02-scheduler-placement/
fixtures/plan.json"
go test -count=1 -json ./internal/scheduler \
  -run 'TestConcurrentSchedulerAttemptsAssignTaskOnce|
TestConcurrentRunOnceAssignsTaskOnceAcrossSchedulers|
TestRunOnceAccountsForNonTerminalAssignedTasks'
```

The actual edition runs passed: model command and concurrency command. The model records three expected assignments, 24 distinct node/task order pairs, the lexical tie result and unchanged pending input states. Its cumulative accounting is b=500m/192Mi and c=800m/256Mi, including b's existing task. It also rejects duplicate node/protocol/port reservations and any offline assignment.

The permutation test uses all six node orders and four distinct task orders, including reversal. It compares complete assignment outputs, including the expected update timestamp and allocated ports. A separate equal-score fixture removes existing work and gives one dynamic candidate two identical ready-node choices: b wins lexically. These are finite input checks, not a proof over every possible cluster state.

The upstream concurrent tests use shared fake-store state. One invokes competing assignment persistence calls; another uses a barrier so two independent scheduler instances both read the same pending task before proceeding. Their oracles require exactly one returned claim, an assigned stored task and one assignment event. The successful run does not create two MemoryService servers and label them an HA cluster.

Timestamp guards and the database track

An assignment carries the task snapshot's ExpectedUpdatedAt. The store's update requires the same timestamp and pending status, so a second stale writer cannot successfully claim that same row. This is narrower than reserving all CPU and ports atomically across several different task rows. Two planners choosing different tasks can each win its own claim while relying on overlapping capacity snapshots. The local plan's accounted budget does not eliminate that race.

The inspected PostgreSQL path adds `FOR UPDATE SKIP LOCKED` for the transaction-scoped read and performs the conditional update. The exact getTask and AssignTask code defines the behavior; the presence of a row lock should not be generalized to the entire resource-allocation problem.

The database observation is shared with Lab 08, which owns a labelled disposable PostgreSQL container and fresh recovery databases. Its full-schema record observed one winner and one conflict after all 16 migrations. The untouched upstream store suite separately had two failures because its reset fixture applied only migrations 1–14 while current node queries need the column added in 16. Keep that failure visible; a passing teaching fixture does not repair or rename the upstream suite. Do not execute the reset helper against a shared database. The fresh whole-book run records database work under Lab 08 rather than pretending the same execution occurred twice independently.

Trace the resource boundary

Boundary	Observation or source	Confidence and limit
Declared requests and limits	Canonical fixture	Exact input values; not utilization
Plan accounting	Three assignments and cumulative budget	Executed finite model, requests/ allocatable
Registration and heartbeat	Actual MemoryService methods in teaching program	Explicitly supplied advertised values, no physical host measurement
Immediate service creation	600m/256Mi request assigned to a red 100m/ 64Mi node	Executed bypass comparison; no container started
Agent task response	Ordinary task lacks ResourceRequirements	Executed inspection of actual method result
ContainerSpe c construction	Agent omits Resources	Inspected caller; creation fields
Actual Docker configuration	Lab 01 resource comparison	Executed owned containers: ordinary zero limits, direct supplied adapter limits

The bypass comparison deliberately advertises a tiny node and submits a larger service with a mismatching label. MemoryService immediately assigns its task. It does not physically stress a host or change the pinned implementation. The real Docker comparison is in Lab 01; its request, inspected HostConfig and cleanup are distinct from this fake-state planning experiment.

Two documentation mismatches affect the worksheet. SCHEDULER.md's limitations still say there is no port or volume awareness, although current Plan filters both. RESOURCES.md says requests remain counted until actual terminal status, but IsActiveTask excludes desired stopped/ removed immediately. The program tests that exception explicitly. A physically running container can therefore outlive its accounted

reservation. The no-overcommit assertion applies to the supplied accounted model, not every physical process during stop acknowledgement delays.

Exercises and worked decisions

1. Why does atlas-01 select c? The fixed port makes b ineligible before scoring; a is offline. More free memory cannot rescue an ineligible node.
2. Why does atlas-02 then select b? Its dynamic port fits and b has 448Mi free versus c's 384Mi before placement. Limits are not the requested budget.
3. Can atlas-06 share c with atlas-01? No: the same TCP31080 is already reserved on c, while b also has that reservation. Another protocol is a different key.
4. What does one competing claim prove? Only one stale pending-task snapshot became an accepted assignment in the tested shared store. It proves neither durable server composition, global reservation, leader fencing nor cessation of a partitioned worker's side effects.

Model cleanup is process exit and removal of only the copied teaching directory if desired. Keep the fixture and execution records. Lab 08 alone owns database cleanup; Lab 01 records exact container cleanup. Never replace those owners with a daemon-wide prune, shared-schema reset or termination of unrelated processes.

Lab 03 — Reconciliation and ownership under drift

Atlas has three separate questions: how many task records should exist, which local containers belong to this agent, and whether the control plane will accept the agent's next status report. Predict each answer before running the fixtures. A container that disappeared is not sufficient permission to reset a running task to `pulling`.

This lab is an original **teaching extension** against source `cbfc94ade5b57e449d6b73bf1485b1c5b1647022`. Its observed track uses upstream `FakeRuntime`, the actual in-process `MemoryService` lifecycle, and the separate store-backed reconciler with its upstream test store. It uses no HTTP listeners, Docker daemon, PostgreSQL database, or external images. Source files copied for private-package access remain byte-identical; the only additions are two test files. The shipped server does not start the store-backed reconciler.

Predict the two passes

Start with the five intent cases in fixtures/cases.json. Write the expected new task count, ordered stop directives, service status, and transition-event count after two unchanged passes.

Case	Initial state	First directive	Second unchanged pass
Below target	Three replicas, one running task	Create two pending records	No additional records/events
Failed, always	One replica, one failed record	Create one pending replacement	No additional records/events
Failed, never	One replica, one failed record	Preserve failed record, no replacement	Same failed record

Case	Initial state	First directive	Second unchanged pass
Scale order	One desired replica, running + pending-a + pending-z + assigned	Stop pending-z, pending-a, assigned	No additional stop/events
Soft deletion	Deleting service, one running task	Desired stopped, actual still running	Service remains deleting

The last case then receives explicit removed-task observations in the store fixture. Only the next pass marks the service deleted. This fixture is not a physical runtime acknowledgement transport. It isolates the reconciler's decision after that observation becomes available.

The exact stop ordering prioritizes lifecycle state, then descending task ID. Desired stopped tasks stop counting as active immediately; this does not prove that their physical containers have already stopped. Deletion logic has the stronger terminal removed requirement.

Run the isolated fixtures

Prerequisites are the locked Go 1.25.0 toolchain, its cached dependencies, Python 3, and a disposable copy of the locked application source. Do not execute against `evidence/orch`. The runner verifies all 361 baseline file hashes before making its two private `.book-labs/lab03*`

package overlays. It never edits baseline files. Set the paths to your actual source copy and Go installation; the example values describe the generation environment.

```
python3 books/inside-orch/scripts/run_recorded.py \
  --id lab03-reader-run \
  --cwd /home/popac/alekpopovic.github.io \
  --output validation/labs/lab03-reader-command.json \
  -- python3 books/inside-orch/scripts/run_drift_lab.py \
  --source-copy /tmp/inside-orch-labs-cbfc94ade5b5/orch \
  --go-root /tmp/inside-orch-gomod/golang.org/toolchain@v0.0.1-
go1.25.0.linux-amd64 \
  --output-prefix validation/labs/lab03-reader-observed
```

Use fresh output names: existing evidence is deliberately not overwritten. `run_recorded.py` captures UTC start/end, command, working directory, exit code and raw stdout/stderr. `run_drift_lab.py` copies unchanged agent/reconciler package and test files, adds the original tests, and executes only `TestBook*` with `go test -count=1`. The runner's fixed cache paths are generation defaults; adapt those environment locations for another host. The canonical tests contain executable fixture setup; the JSON fixture file describes and checks the exact case ID set.

Observe lifecycle and ownership together

The ten agent cases all include an unmanaged container with a similar name and a managed container belonging to another node. Both must survive **each of the two passes**, including all cleanup cases. Matching uses labels, never name similarity. Exact ownership predicates distinguish adoption from cleanup eligibility: matching additionally requires service, version and image consistency; cleanup eligibility requires managed, matching node and matching task identity.

Case	Observed first-pass result
Missing, assigned task	Pull/create/start replacement; task becomes running
Missing, running task	`pulling` report rejected; no replacement created

Case	Observed first-pass result
Stopped, matching owned container	Start the existing container; report running
Image mismatch, running task	Remove owned mismatch, then reject `pulling`; no replacement
Version mismatch, assigned task	Remove owned mismatch and create replacement
Service-label mismatch, running task	Remove owned mismatch, then reject `pulling`
Foreign-node referenced container	Preserve it; reject attempted `pulling` reset
Unexpected owned orphan	Stop/remove orphan; unknown-task removed report is ignored by cleanup
Unmanaged referenced lookalike	Preserve it; reject attempted `pulling` reset
Soft-deleted owned task	Remove owned container; accepted removed report finishes service deletion

The precise UUID-bearing rejection is retained per case in the actual check. Its stable suffix is `cannot accept agent status "pulling"`. The running task remains running in control-plane state. This is a **passing test of an observed failure boundary**, not a successful recovery claim. Upstream agent unit tests use a more permissive fake client; connecting the unchanged agent to actual `MemoryService.ReportTaskStatus` exposes the invalid running-to-pulling transition.

Inspect currentTaskContainer, then ensureTask. Ordering matters: cleanup of an owned mismatch can occur before the later rejected lifecycle report. The lab leaves that result visible rather than modifying the agent or allowing an illegal transition. A second unchanged pass must add no transition events; repeated drift observations and rejected report attempts may still occur and are not transition-event duplicates.

Optional reader Docker track

This track was **not executed for this lab**. Use a disposable host, inspect socket authority and `docker info`, and read the pinned runtime integration test before opting in. From the disposable application copy, the actual gate and test target are:

```
ORCH_DOCKER_INTEGRATION=1 go test -count=1 -run
'^TestDockerRuntimeIntegration$' ./internal/docker
```

The daemon can grant host-level authority. Confirm the test's ownership labels, capture each created container ID, and retain the cleanup result. Clean only those captured IDs or an equally strict lab-owned label set. Never run broad container deletion or target another node's containers. A Docker runtime test still does not run this lab's MemoryService/agent lifecycle comparison; the fake track and daemon track establish different facts. The book's separately recorded real Docker flow is in Lab 01, with its own captured IDs and cleanup.

Explain the boundary

Question	Owner	Evidence in this edition	Remaining limitation
Desired replica task intent	Store-backed reconciler	Five actual cases, repeated no-op pass	Not composed into default server; fake store is not SQL
Local managed-container drift	Agent plus runtime and report contract	Ten actual FakeRuntime/ MemoryService cases	No daemon behavior proven; running-to-pulling rejects recovery
Process restart	Binary lifetime and state owner	Separate Lab 01	Default active state is memory; container survival does not restore records

Question	Owner	Evidence in this edition	Remaining limitation
Database restoration	Migration/ backup/store composition	Separate Lab 08	Restoring tables does not wire the default binary to persistent state

Explain why changing only the fake client would hide the most useful finding. Propose a separate recovery design with legal lifecycle transitions, retry behavior and failure ownership; label it design work until tested. Do not patch the immutable baseline. Cleanup for this track is process exit plus the two owned overlay directories; it creates no external containers or database objects.

Lab 04 – Rollout, Rollback and Modeled Availability

Atlas has three useful replicas. The fourth slot is deliberately temporary: a new version must earn a healthy report before the controller can ask an old replica to stop. In this lab you observe the real rollout controller and scheduler against MemoryService, then separate their arithmetic from application traffic. No Docker daemon, image pull, database or external endpoint is used. Agent reports are synthetic inputs, explicitly supplied by the teaching harness.

Read Chapters 10, 13, 14 and 17 first. Use the pinned source and Go 1.25 toolchain described in the lab environment. The complete teaching extension is additional book code, not a patched application. Its service and node IDs are generated by the unchanged store; captured IDs and UTC timestamps identify this particular execution. Logical names, action order, CPU quantities and assertions are fixed. Do not describe the random identifiers or elapsed timings as deterministic.

Inputs and Cleanup

The baseline creates one node with CPU 300 milli-units and memory 1 GiB. Three service tasks each request CPU 100 and memory 1 MiB. Their image labels are nginx:1.27-alpine and nginx:1.28-alpine, but neither image is pulled. A restart-never policy keeps injected failure distinct from automatic replacement. The rollout uses maxUnavailable zero and maxSurge one. Health is an explicit report, not inferred from running.

Before running, verify all copied baseline files against source-lock.json. Copy the extension into .book-labs/rollout inside a disposable Orch module. Keep the original snapshot untouched. The process owns only in-memory objects and its selected CSV output. Cleanup is process exit and, when no longer needed, removal of that exact copied extension directory. Never remove a real service or container to imitate a synthetic failure.

From that disposable module, with the explicit pinned GOROOT, caches and GOTOOLCHAIN=local selected:

```
GOFLAGS=-buildvcs=false go run ./.book-labs/rollout /tmp/inside-
orch-rollout-timeline.csv
```

The command writes a CSV and prints a JSON observation document. Choose an output file owned by this run. Save both with the actual command, tool version, source SHA, UTC times and exit code. The author's recorded command, raw observations and timeline are existing execution evidence; reviewing them is different from executing a new run.

Predict Each Pass

First predict whether a pass creates, waits, stops, fails or completes. The first controller pass changes the deployment from pending to running and creates one pending target. That fills the modeled four-task ceiling. The scheduler cannot assign it because the three original requests consume all advertised CPU. An additional controller pass must retain the same target rather than create a fifth task.

The fixture then explicitly changes advertised node capacity to CPU 400. This is a controlled input change, not a claim that physical hardware expanded. The scheduler assigns the pending target. A running report still leaves all three old replicas active: running is insufficient for target readiness. A healthy report authorizes one stop directive, leaving three modeled available tasks. The harness acknowledges each old stop with stopped then removed before advancing the next cohort.

The captured baseline has seven checkpoints: requested; target-created; capacity-exhausted; running-but-not-healthy; and three healthy-controller passes. For each, inspect old_active, target_active, target_healthy, pending_stop, modeled_available and surge_bound. Add old_active and target_active: the sum must not exceed four. Whenever old_active decreases, enough targets must already be healthy to satisfy the stop gate. Do not substitute the count of all stored historical task objects for active work.

The final observed checkpoint is succeeded while one final stop directive is still unacknowledged. That is an especially useful boundary. The active predicate excludes a task whose desired status is stopped, although its last actual report may still be healthy. Deployment success here is the controller's model; it is not proof that Docker removed every old container. The harness deliberately retains that observation rather than retroactively calling it physical cleanup. There were no containers in this track.

Failure and Conflicting Operations

The harness repeats an identical active rollout and asserts the same deployment ID. A changed image conflicts. Scale during the rollout conflicts, and an active rollback also conflicts. These outcomes are actual MemoryService operation checks, not invented CLI force flags. A stale deployment status write using the original UpdatedAt is rejected after the controller advances the object. That is the store's optimistic concurrency guard. Agent task reports have no equivalent caller-supplied timestamp field; the separate late healthy report tests terminal-state rejection instead.

A fresh fixture creates a target, assigns it explicitly and reports failed. This assignment is an injection seam, not scheduler proof of available capacity. The controller marks the deployment failed. A later healthy report cannot revive that failed task. Rollback creates a distinct rolling_back deployment from version two to version one and preserves the failed deployment's ID, completion timestamp and status. Version one qualifies through the initial-baseline special case; no previous version on an untouched service produces invalid state. A separate fixture verifies that delete cancels active deployment work by marking it failed.

Compare these observations with the unchanged operation_lock_test.go, rollback_test.go and rollout_test.go source. The executed package selection also includes these fixtures. Maintenance windows and an audited emergency bypass are separate policy checks in operations_test.go. The passing

TestMaintenanceWindowAndForcedBypass does not make force a normal rollout requirement. Do not invent pause/resume, an automatic rollback or a progress deadline: the baseline does not provide those controls.

Worked Questions

1. **Why does full capacity wait safely?** The controller has already consumed the one surge slot with a pending task. The scheduler finds no CPU fit. No new healthy target exists, so the old-task stop gate refuses progress. Safe waiting may be indefinite; an operator must diagnose capacity or health rather than assume a deadline will recover it.
2. **Does maxUnavailable zero prove zero downtime?** No. It constrains reported task counts in this implementation. A traffic claim additionally needs owned endpoints, a real probe stream, timestamp resolution, sampling intervals, response criteria and routing convergence observations. None were collected here.
3. **Why retain failed history when rolling back?** It preserves the failed decision and its terminal evidence while a new deployment owns recovery. Rewriting the failed record to success would destroy the distinction between the incident and its response.
4. **What survives a controller restart?** Reconstructing a controller over the same store retains objects. A fresh default server constructs a new MemoryService and loses them. Durable restart requires a separately integrated persistent store and new-process test; neither an in-memory timeline nor the existence of PostgreSQL migration files establishes that composition.

The machine acceptance record checks the captured budget, health gates, negative operations and terminal history. It certifies this controlled execution scope, not external availability or production readiness.

Lab 05 – Secrets, Identities and Probe Boundaries

Atlas needs a secret without turning every metadata response into a secret-distribution channel. Its worker also needs an identity distinct from an operator's role. This lab exercises those boundaries using the unchanged local envelope, API handler, credential issuer and health checker. It creates no Docker workloads, queries no public host and uses no real credentials. The complete teaching harness generates the key, signing material, bootstrap token and secret canary in process memory. Raw values are never written to the evidence files.

Read Chapters 14, 23, 24 and 27 first. Use the environment contract, the pinned Go 1.25 compiler and a verified disposable source copy. This is a teaching composition of actual packages, not a deployed production identity system. HTTP API requests run directly through httptest recorders. The only sockets are two disposable loopback healthcheck listeners, automatically closed on exit. If local listeners are unavailable, report the probe track blocked; do not silently replace observations with predictions.

Reproduce Without Publishing Secrets

Copy the harness into .book-labs/security under the disposable Orch module after verifying its baseline files. Select a fresh evidence directory under your own temporary workspace. Do not direct output into another person's logs. With the environment's explicit pinned GOROOT and run-owned caches:

```
GOFLAGS=-buildvcs=false go run ./.book-labs/security /tmp/inside-orch-security-channels
```

The program prints a sanitized result and writes only channels already scanned for the generated canary. Capture its exit code and command independently. Its actual successful command and case matrix

document the author's run. Earlier attempts remain recorded: one exposed an invalid raw zero-resource JSON fixture, another assumed which of two randomly identified nodes would receive a task, and one was blocked by sandbox socket permissions. Correcting a teaching fixture is different from repairing the immutable application. The final fixture registers the intended assigned worker before creating the secret-bearing service.

The raw HTTP body uses empty resource request/limit objects where no quantity is intended. Directly marshaling a zero-value Go Resources struct emits cpu:0, which the strict quantity decoder rejects. The public CLI client's normalization avoids that boundary. This is why a source type and an accepted wire document must be checked separately rather than assumed interchangeable.

Cleanup closes the owned listeners and discards in-memory state. There are no host ports to retain, database schemas to reset or containers to remove. Delete only this run's temporary evidence and copied harness when finished. Keep sanitized hashes needed for review. Never place a real API token, database URL or envelope key in these examples.

Follow Sensitive Data Across Owners

The harness creates a one-use canary and encrypts it with LocalEnvelope. A same-key round trip must recover the exact value; a different key must reject decryption. The ciphertext is scanned to ensure it does not literally contain the canary. The result contains the canary's SHA-256 and ciphertext hash, not their plaintext. These observations exercise this implementation, not cryptographic certification or a remote key-management service.

The API creates a secret in team-a, then returns metadata for create, get and list. Events, audits, captured logs and a cross-namespace missing-reference error are forbidden plaintext channels. Each is scanned before persistence. The scanner records file bytes and hashes so the acceptance script can detect later changes. Merely searching a final

JSON report would miss a leak in an intermediate response; conversely, searching every payload without a visibility policy would incorrectly classify intended agent delivery as a leak.

An authenticated assigned worker receives the canary in its task environment, verified by an in-memory equality comparison. That raw response is discarded. The same secret name requested from team-b is hidden, and a service in team-b cannot resolve team-a's reference. This demonstrates namespace-scoped lookup, not kernel or network isolation. The server decrypts the value and the agent/application must receive it; anyone controlling those trusted processes can cross the confidentiality boundary.

Producer	Consumer	Permitted sensitive data	Enforcement point	Residual assumption
Secret writer	API/server	One-use plaintext input	User role and selected namespace	Transport and server process trusted
Local envelope	MemoryService	Ciphertext and key identity	AES-GCM/key identity check	Key remains available and secret
Metadata API	Viewer/ operator	Name and timestamps	Metadata projection	Other fields do not become covert leaks
Server assignment	Authenticated assigned agent	Plaintext environment	Issued node credential and task node ownership	Worker host and process trusted
Agent/ runtime	Application	Required plaintext environment	Intended workload setup	Docker socket owner has host-level power
API/ handlers	Logs, audits, errors	Sanitized metadata	Channel-specific redaction and canary checks	A finite scanner does not prove every future log safe
Health checker	Local workload endpoint	Probe request, no canary	Input path validation and same-origin redirect guard	Chosen address/DNS require their own policy

Losing the envelope key makes existing ciphertext unusable; changing a secret is not the same as rotating the encryption key. MemoryService ciphertext also disappears with the process. Encryption protects one storage representation while durable recovery requires an independently designed state and key lifecycle.

Separate User Authorization From Worker Identity

The matrix observes 401 for a missing required user token, a successful viewer read, 403 for viewer mutation, an allowed team-a operator create and 403 for that operator in team-b. Audit reads require cluster administration: a namespace operator is denied, while an admin succeeds. Tokens are signed using the source's SignJWT interface with synthetic signing material, not hand-written strings that imply external identity-provider support.

Worker registration without the configured bootstrap token returns 401. Registration with it issues a credential. A second worker's credential cannot report status for the first worker: the handler returns 401 and the task state remains unchanged. A successful heartbeat rotates the first credential; replaying its old credential returns 401. The issuer separately rejects a token exactly at its expiry boundary and rejects a revoked record. These are distinct from user-role failures and therefore must not be flattened into a universal 403 expectation.

The empty user-JWT configuration is tested explicitly: an anonymous service metadata read succeeds. Agent credential enforcement is a separate path. The optional user JWT mechanism, static bootstrap credential and rotating node bearer tokens do not provide mutual TLS, public identity lifecycle integration or containment after a Docker socket compromise. A lost rotation response is also an operational concern; passing rotation tests does not establish a replay-safe recovery protocol.

Keep Probe Traffic Local

Two loopback listeners provide a legal same-origin redirect, a cross-origin redirect, a slow response and a body larger than 64 KiB. The same-origin request succeeds. The cross-origin request is unhealthy and the second listener's request count stays zero. The slow endpoint exceeds a ten-millisecond timeout. The large response succeeds while the checker limits body consumption to its source constant; that observation does not claim the server wrote only 64 KiB or that allocation is globally bounded.

Input validation separately accepts /ready and rejects both a full absolute URL and a protocol-relative path. Those rejected examples never cause DNS or network access. The checker receives an already constructed target; validating a relative path and blocking redirects does not constrain every possible initially selected address or DNS answer. Do not probe cloud metadata or a production private service to illustrate that gap.

Worked Questions

1. **Why is plaintext assignment allowed?** The trusted worker needs the application's value. Its credential and task ownership authorize a narrower distribution boundary than public metadata reads. The harness proves equality without saving the raw payload.
2. **Why distinguish 401 and 403?** Missing or wrong credentials fail authentication. A recognized viewer attempting a mutation fails authorization. Exact handler mapping determines the result; a generic denial label loses that evidence.
3. **Does zero cross-origin traffic prove SSRF safety?** It proves this redirect was blocked before contacting its target. Initial address choice, DNS resolution and trusted network placement remain separate questions.
4. **What does a clean canary scan establish?** The selected captured channels omitted this one generated value and still match recorded

> hashes. It does not prove universal redaction, absence of derived sensitive data, durable encryption or production-safe deployment.

The acceptance record checks the actual matrix, unchanged task, probe destinations and persisted channel hashes. Preserve those boundaries when applying the lesson to a live system.

Lab 06 — Namespace boundaries and competing quota admissions

By Aleksandar Popovic. Source edition: `cbfc94ade5b57e449d6b73bf1485b1c5b1647022`.

Problem and experiment boundary

Atlas Payments and Observability both need a service named `api`. An operator must distinguish namespace lookup, authorization, resource admission and physical isolation. This executable teaching extension constructs the actual API handler and MemoryService, sends real HTTP request objects through them, and records generated object IDs. It uses `httptest.NewRecorder`, without binding a socket. There is no connected host, Docker daemon or PostgreSQL request in this lab. Injected JWTs use a synthetic process-local key and are never printed.

Prerequisites are the locked disposable Orch source copy, Go 1.25.0, cached locked dependencies and the book fixtures. The previous lab's security observations inform the permission predictions; its process state is deliberately not reused. This lab initializes its own namespaces and quota state. Lab 07 likewise initializes independent state and carries the namespace/admission lessons forward rather than inheriting these objects.

Inspect `internal/namespace/context.go`, `internal/api/namespace_test.go`, `internal/controlplane/namespace_test.go`, both quota test files, `internal/quota/quota.go`, store locking methods and migrations 000011/000012. Consult `docs/MULTI_TENANCY.md` as documentation and compare it with the selected namespace implementation.

Inputs and setup

`fixtures/tenants.json` describes distinct labels for identifying the scenario. Those labels are fixture annotations: ServiceSpec has no arbitrary Labels field. The actual service uses supported non-sensitive Env markers for the team. `fixtures/api-base.json` supplies one replica, 100 millicores, 64 MiB and one positive public-port declaration. `fixtures/quota.json` sets all seven limits:

Limit	Value	Measured object
Services	2	Active services
Replicas	3	Desired service replicas
CPU	300 millicores	Request times desired replicas
Memory	201326592 bytes	3 × 64 MiB
Public ports	2	Positive declarations in ServiceSpec, once per service
Secrets	1	Namespace-local secrets
Registry credentials	1	Namespace-local credentials

Never run the program in `evidence/orch`. Copy the canonical `examples/teaching-extensions/lab06/main.go` into `.book-labs/namespaces-quotas/main.go` under a verified disposable export of the pinned module. The recorded execution used `/tmp/inside-orch-labs-cbfc94ade5b5/orch`. From that copy run, with your toolchain environment configured:

```
go run ./.book-labs/namespaces-quotas /absolute/site/books/inside-orch/labs/06-namespaces-quotas/fixtures
```

The exact executed argv, working directory, UTC interval and exit code are in `validation/labs/lab06-canonical-fixtures.json`; stdout contains all 28 structured observations. This command has already executed with Go 1.25.0; the placeholder absolute path above is reader setup, not an additional observed run. Export `GOTOOLCHAIN=local`, use the verified Go

1.25.0 `GOROOT`, and set `GOFLAGS=-buildvcs=false` for the exported source tree without `.git`. The standard recorded runner can retain a new run using a new output name. It refuses overwriting prior evidence.

Predictions and observed results

Predict the response before reading `06-check.json`:

Question	Prediction from source	Actual scoped result
Can both tenants create `api`?	Yes, distinct generated IDs	Passed; IDs captured
Can Payments fetch Observability's service ID?	404, including cluster-admin actor with selected Payments scope	Passed
Which selector wins?	`X-Orch-Namespace` before query; query fallback before default	Passed; documentation's header/default shorthand is incomplete
Does a viewer cross scopes?	Own scope 200, other scope 403	Passed
Can namespace admin list cluster namespaces?	403; cluster admin 200	Passed
Do same-name secrets share data?	Independent; cross-scope reference rejected	Passed; plaintext absent from output
Does denied scale mutate desired replicas?	No; scale 1→4 exceeds replica quota	409 structured `quota_exceeded`, replicas stay 1
What happens to two scale requests competing for one slot?	One succeeds, one quota denial	Passed; final total exactly three replicas
Can quota zero mean no replicas?	No; zero means unlimited	1000 desired replicas admitted in separate no-node fixture
Is namespace deletion always possible after service deletion?	No; retained history can block it	Passed; 409 retained and explained

Question	Prediction from source	Actual scoped result
Do independently constructed memory servers share a tenant?	No	Second handler returns 404 and only has default namespace

The arithmetic starts at one service × one replica: 100 millicores, 67108864 bytes, one public-port declaration. Adding a portless worker gives two services and two replicas. Two concurrent updates each request a second replica on a different service. Exactly one update is admitted: three replicas, 300 millicores, 201326592 bytes, one public-port declaration. This proves the in-memory admission operation's tested atomicity, not PostgreSQL serialization. Quota error `Used` and `Requested` represent projected totals in this implementation; they are not old usage and an incremental delta.

Deleting the worker releases its desired-resource accounting. The secret and registry tests separately exceed their one-object quotas and check structured errors without recording their values. Conflicting usage query and selected header returns 404. The default namespace remains undeletable. No test reaches into internal maps to erase history.

Independent assertions and troubleshooting

From the site root:

```
python3 books/inside-orch/scripts/verify_namespace_workload_labs.py
--lab 06
```

This offline verifier checks the frozen input/output hashes, exact source pin, response records, arithmetic and required cases. It validates retained observations; it does not repeat the experiment. A fresh program run produces new IDs and timestamps and must be retained separately. If a quota denial is unexpectedly absent, inspect current projected usage and the selected namespace before changing limits. If a scope test fails, inspect role claims and header/query precedence before assuming object corruption.

PostgreSQL is a separate prerequisite and storage experiment. Require a dedicated `_test` database, verified lab-only container labels, loopback address, all sixteen migrations and exclusive ownership before testing transaction/advisory-lock quota admission. The upstream test reset applies migrations 1–14 and omits `agent_version` from migration 16; root intake retains the observed 42703 failure. Do not inherit this lab's success into a PostgreSQL status. `06-check.json` explicitly says PostgreSQL quota execution was not performed.

Cleanup, rerun and architecture review

All objects are process-local. The harness attempts supported deletions, observes history preventing namespace deletion, then discards the process. It touches no shared database or daemon. Rerun by starting a fresh process, not by forcibly clearing namespace history. Synthetic values never become production credentials.

Boundary	Implemented and measured here	Extension still needed
Logical lookup	Namespace-qualified names and hidden cross-scope IDs	Review every new route for selector propagation
RBAC	Scoped viewer/admin and cluster administration	Hostile tenant threat model and identity lifecycle
Admission	In-memory atomic desired-resource accounting	Separately executed persistent transaction path
Runtime isolation	No claim from namespace metadata	Network, daemon, filesystem and workload isolation
Failure domains	Independent servers do not share active memory	Explicit shared state and recovery integration
Durable recovery	None supplied by this memory harness	Backup, restore, keys and tenant-history procedures

A namespace is useful organization and an authorization boundary within the tested API. It does not create a separate Docker daemon, network or durable control plane.

Lab 07 — Git intent, job attempts and scheduled occurrences

By Aleksandar Popovic. Source edition: `cbfc94ade5b57e449d6b73bf1485b1c5b1647022`.

Problem and scope

Atlas wants Git to describe services, a report to finish once, and a daily report to follow Belgrade wall time. These are three different controllers with different failure windows. This teaching extension runs real local Git, the locked CLI parsers and actual control-plane methods. It injects task reports, a log transport, a clock and a CronJob store. It launches no container, database or network listener. A successful exit-zero report proves the state transition after a report, not execution of the shell command in Docker.

Prerequisites: a disposable verified copy of the pinned module, Go 1.25.0, Git and IANA timezone data. Read `internal/gitops/gitops.go`, `internal/controlplane/workloads.go`, `internal/cronjobs/cronjobs.go`, `internal/cli/workloads.go`, `pkg/types/workloads.go` and their tests. Compare `docs/GITOPS.md`, `docs/JOBS.md` and `docs/JOBS_DESIGN.md` with implementation. The lab initializes `book-workloads`; no previous lab process is assumed to survive. Its connection to Lab 06 is the namespace/admission contract, not retained objects.

Canonical fixtures and parser fidelity

`10-api.yaml` and `20-worker.yaml` are service manifests read through the actual deploy parser in lexical order. Their zero replicas avoid claiming a connected runtime. Both specify 100 millicores and 64 MiB. The Job fixtures include `kind`, name, image, command, backoffLimit and resource strings; the successful job has zero retries and the failing job has one. `cron.yaml` uses the actual CLI envelope: `schedule`, `timezone`, `concurrencyPolicy`, and `jobTemplate` image/command/resources. It

deliberately omits unsupported CLI fields such as missedScheduleLimit or jobTemplate.backoffLimit. Direct controller fixtures set missedScheduleLimit separately and are labelled in-process tests.

The harness invokes the actual CLI root with an injected capture client. The parser still requires `--server`; the supplied `http://127.0.0.1:1` is never contacted because the capture client implements the called methods. The first run exposed this missing setup and failed. Its record remains `validation/labs/lab07-gitops-jobs-cron.json`; the corrected implementation adds the required CLI option. The final canonical run passed all 26 cases.

Copy `examples/teaching-extensions/lab07/main.go` into `.book-labs/gitops-jobs-cron/main.go` in the disposable pinned module, then run:

```
go run ./.book-labs/gitops-jobs-cron /absolute/site/books/inside-orch/labs/07-gitops-jobs-cron/fixtures
```

Use Go 1.25.0, `GOTOOLCHAIN=local`, its verified `GOROOT`, and `GOFLAGS=-buildvcs=false` for the source export. The exact executed argv, cwd and UTC interval are retained in `validation/labs/lab07-canonical-fixtures.json`, with stdout containing generated source/job IDs and full local commit SHAs. The illustrative absolute path is reader configuration. A new run must retain new records rather than overwrite the previous run.

GitOps experiment

The program creates a uniquely owned temporary local repository, disables hooks for its Git commands, and commits only synthetic manifests. The actual CommandFetcher clones that local repository; no Git operation touches the immutable evidence repository. Fetch unavailability is separately injected and never uses a fake external host.

Action	Prediction	Actual observed result
Initial sync	Managed names in lexical order; recorded full revision	`atlas-api`, `atlas-worker`; source ID and revision captured
Unchanged sync	No false version increment	Passed, version stays 1

Action	Prediction	Actual observed result
Manual scale in warn mode	Desired/live drift visible, manual replicas retained	Drifted with replicas 2
auto_revert status request	Desired replicas restored	Replicas 0; status call mutates state
Failed fetch	Existing objects retained and no prune	Both services remain active
First valid file, second invalid	Earlier application can survive later failure	API replicas 1, worker retained, LastRevision unchanged
Remove managed worker, sync	Worker deleted, unrelated manual service retained	Passed
Credential-bearing repository URL	Rejected without synthetic private value	Userinfo and token-query cases passed

The recorded fixture revisions are real commits created by that execution, not the application source SHA. Sequential apply is not a transaction: a failed sync can leave an earlier manifest applied even though LastRevision still names the previous success. Prune uses previous managed names within the namespace; preserving a never-managed object does not establish safe ownership transfer for a reused managed name. The auto_revert status path restores recorded desired state directly; it must not be described as a new admission decision or a passive read.

Jobs and latest-attempt logs

The successful parser fixture produces requests/limits of 100 millicores and 1048576 bytes and an explicit shell command. A real MemoryService creates an owned task, receives injected running/stopped reports with exit code zero, and marks the job succeeded. No process executes the command here.

For the failing fixture, BackoffLimit 1 means an initial attempt plus one retry. The first injected nonzero stop creates a new attempt task and pending state with Attempts 2; the second ends failed with Attempts 2. The CLI log command selects the latest attempt task ID. Its transport is an explicit fake that writes a marker to an in-memory/discarded destination; this is argument-selection evidence, not captured report output. No timed backoff, jitter, deadline, parallelism, completion count, TTL or HA uniqueness is inferred.

Cron decisions and daylight saving

An injected clock advances from `2026-08-21T12:00:00Z` to `12:03:00Z`. With one-minute schedule and missed limit 2, Allow creates two occurrences and advances the checkpoint to `12:02:00Z`. Forbid with an already-active job creates none and leaves its checkpoint unadvanced. Replace deletes the active job metadata and creates one replacement. Suspension creates none; resuming catches up two retained occurrences. These are actual controller/store calls with an injected clock and store, not a daemon waiting until those dates.

Belgrade scenario	UTC base	Actual computed next/due instants
Daily local 02:30 across missing spring hour	2026-03-28 23:00Z	Next 2026-03-30 00:30Z; March 29 02:30 does not exist
Daily local 02:30 across repeated fall hour	2026-10-25 00:00Z	00:30Z at +02:00 and 01:30Z at +01:00

The parser accepts exactly five cron fields. The implementation matches all specified fields, including day-of-month and day-of-week. Job creation and CronJob checkpoint update are separate operations; a crash between them can repeat an occurrence. A future durable uniqueness key such as `(cronjob_id, scheduled_instant)` would require an atomic storage design and executable recovery test. It is not supplied by this lab or the default memory server. Forbid's behavior is measured with a job already active at tick start; it does not certify one job across every catch-up loop.

Assertions, cleanup and rerun

```
python3 books/inside-orch/scripts/verify_namespace_workload_labs.py
--lab 07
```

The offline verifier checks retained logs and canonical hashes, all required cases, real revision forms, latest job outcomes, preservation of unrelated services and exact timezone results. A new Go run is the fresh experiment; the verifier alone is not. On failure inspect the recorded parser envelope or UTC instant before changing expected values.

The temporary fixture repository is removed by the program, GitOps checkouts self-clean, jobs are removed through supported methods, and memory state is discarded. The harness never deletes shared containers, databases or source evidence. Rerun starts a fresh temporary repository with new commits and IDs.

Mechanism	Implemented observation	Durability or safety boundary
Source-of-truth reconciliation	Local Git fetch, warn/status revert, name-based prune	Partial apply; same-name ownership and re-admission gaps
Finite execution	Owned attempts and report-driven terminal state	Injected reports; no Docker execution or persistent job recovery
Scheduled intent	Five-field parser, overlap choices, bounded catch-up, real timezone computation	Separate occurrence creation/checkpoint; no exactly-once promise
Durable occurrence deduplication	Not supplied	Requires a separately implemented transaction/uniqueness extension

Lab 08 — Recover artifacts and state exactly what was recovered

Recovery starts with an inventory. Desired service specifications, observed task status, event and audit history, encrypted secret rows, envelope keys, Docker images and volumes, process configuration and monitoring data have different owners. A database dump can only contain what was actually written to that database. The default Orch server keeps active orchestration intent in MemoryService. Its recovery boundary cannot be enlarged by a successful backup command.

This capstone has three tracks: source/schema inspection, a bounded fake-agent load measurement, and an actual PostgreSQL restore into a fresh replacement database. It depends on the earlier lifecycle, namespace, security and placement lessons. The author-run records are specific observations, not a promise that any available database or Docker host is safe to reuse.

Source and schema inventory

Inspect all16 up/down pairs under the pinned `migrations` directory. `internal/version/version.go` declares schema16 and accepted range15–16. Discovering a pair means a reversal file exists; it does not prove every downgrade preserves application semantics. The migration manager sorts versions, holds a PostgreSQL advisory lock, executes each up migration and its ledger insert in a transaction, and skips already-applied versions. `Status` creates its ledger table if absent, so it is not a strictly read-only database operation.

The source store fixture applies only1–14. Its actual guarded run failed two tests because current node queries refer to `agent_version`, introduced by migration16. Keep that result alongside the complete-schema teaching experiment; do not repair the immutable fixture or call the original suite green.

`backup-db.sh` requires an explicit database URL and invokes `pg_dump` in custom format. `restore-db.sh` requires a target, an existing input and `--yes`; `--clean` is explicitly destructive. These scripts cannot establish that the target is a disposable book database. This lab's additional target guards reject missing names, source/target equality, names outside the book prefix, existing replacement databases, remote endpoints, wrong container ownership labels and mismatched published ports. Restoring into a fresh target avoids overwriting a live source even when restore itself fails partway through.

The actual disposable PostgreSQL experiment

The book creates a uniquely labelled PostgreSQL16 container with a random loopback host binding and temporary storage. Its ownership guard verifies `inside-orch.lab=true`, a unique run label, the exact configured database name and the `127.0.0.1` port mapping before any destructive suite. Do not change the guard to accept an existing production database.

The recovery orchestrator creates two new, distinct lab database names only after confirming neither exists. It verifies all361 source-copy hashes, invokes the full-schema teaching program, and records every executed argument vector and exit code. The program observes16 pending migrations, runs `Up`, verifies current000016 with none pending, runs `Up` again, and confirms that down without explicit permission is rejected. It does not execute an allowed downgrade in this lab.

The fixture creates one node, one service and one task. Two goroutines compete to assign the same pending task using the same expected timestamp; one succeeds and one receives conflict. This validates a per-task claim on the complete schema. It does not establish global capacity reservations, server leadership, or a durable control-plane composition.

A random one-use secret canary is encrypted and stored. Only its SHA256 digest enters the published record. `pg_dump` runs inside the owned container and the binary dump is hashed. `pg_restore` writes into the fresh replacement database. The verifier checks schema16, service/

node row counts and secret decryption with the same envelope key; the decrypted digest must match. A different key must fail. The key used here is explicitly lab-only teaching material, not owner configuration. Encrypted values are not useful recovery artifacts if their corresponding key material has been lost.

The actual recovery record captures the dump hash, full tool command vectors, source/target names, created object IDs, schema observations, restored row counts and digest comparison. Its cleanup removes precisely the two databases it created. The temporary container remains available for later isolated book checks until the root run's final cleanup; it is not a required public service.

The same program tests two independent PostgreSQL connection pools for an important leadership boundary. After cancellation, `Lease.Done` closes and `Release` returns nil, yet the other pool still cannot acquire the advisory lock. Closing the original pool releases the physical session and the second pool then acquires. This is an observed package defect in the locked release. It is separate from the default server, which does not start this leadership package.

Bounded fake-agent measurement

Use the actual `orch-loadtest` flags: three fake nodes,30 services, two replicas, failure-rate0.02, duration2s and seed20261013. The command record and raw counters preserve the measured run. Record the machine architecture, available CPU count, Go version and source hash alongside these inputs. A fixed pseudo-random seed stabilizes the choice stream; wall-clock loop scheduling and measured durations can still vary.

Read the load generator before interpreting its report. It uses MemoryService directly, fabricates task transitions and sleeps between ticks. Its scheduler/reconciler-labelled counters name operations in this simulator; they are not proof that standalone controller binaries or real Docker containers handled the workload. No PostgreSQL query

throughput, container startup latency or HTTP service-level objective was measured. One small run is sufficient to inspect counters and their definitions; repeat only to answer a stated variance question.

The baseline source's successful fixture sizes and counters describe this bounded simulation. They cannot justify selecting production hardware, publishing a capacity number, or promising a recovery-time objective. For a real service proposal, define start and stop events, complete intended-state recovery, application checks, missing-secret handling, orphan-container review, agent re-enrollment and residual data-loss accounting before measuring a duration.

Verify scope and clean up

`python3 books/inside-orch/scripts/verify_lab_records.py --lab 08 --mode inspection` inventories migration pairs and composition without connecting to a database. Recorded mode validates the author-run migration, restore, target-guard and load evidence and writes `validation/labs/08-check.json`. The report must keep full live orchestration recovery as `unsupported_default_runtime`; a restored-artifact pass is narrower.

Preserve the dump hash, raw measurement output and command records. Remove only owned child databases, exact captured containers and the run's temporary files. Never use source/target equality as a convenient restore shortcut. Test guard rejection before relying on a successful restore: target guard tests reject empty, equal and unrelated names. These tests are teaching safety controls; they do not silently modify the source project's scripts.

Worked recovery decisions

What does an equal canary digest prove? The encrypted row survived dump/restore and the same envelope key can decrypt its original bytes. It does not prove every application secret was included, that keys are backed up securely, or that the default server loaded the restored secret store.

Why are leadership and backups separate? Backups describe past persisted artifacts. A leader lock coordinates current sessions. A complete dump cannot revoke a partitioned worker, unlock a healthy pooled session, or create fencing tokens for remote side effects. Each requires its own mechanism and test.

What must a future durable extension demonstrate? It needs API/interface parity, transactional mutations with events, correct restart hydration, tenant scoping, key continuity, controller reconstruction, remote ownership reconciliation, fencing and an application-level recovery oracle. Reusing the current pgx package is a starting point, not evidence that these requirements are satisfied.

Isolated laboratory contract

Source edition: `cbfc94ade5b57e449d6b73bf1485b1c5b1647022`. Commands below are procedures, not captured results. Actual runs belong in `validation/labs/` with UTC start/end, command, exit status, prerequisites and working directory. The read-only capability report is `validation/intake/lab-environment.json`.

Two tracks

Source-only track: copy the locked source outside the website, build with Go 1.25 or newer (`GOFLAGS=-buildvcs=false` for the source export without .git; set GOROOT to the chosen toolchain, GOENV=off and run-owned caches) and run `go test ./...` with external integration environment variables unset. Scheduler, lifecycle, resource parser, API/FakeRuntime, rollout, namespace, policy and deterministic controller assertions need neither Docker nor PostgreSQL. Tests may open loopback HTTP listeners. An ordinary pass does not mean gated Docker/database tests ran. Fuzz targets are separate timed exercises; build caches and downloaded dependencies belong outside canonical evidence.

Disposable execution track: run separately built CLI/server/agent binaries, then opt into Docker and PostgreSQL experiments. Docker socket access grants broad control of the host daemon; use a dedicated disposable daemon/VM where practical. An agent launched against the regular host socket can affect other orch-labelled workloads, so never use broad adoption/cleanup on a shared worker. Database integration code in `internal/store/postgres_test.go` executes `DROP SCHEMA public CASCADE`; the guard below must identify a dedicated container before invocation. A PostgreSQL server beside `orch-server` does not persist `MemoryService` maps.

Explicit owned resources

Resource	Scope and privilege	Planned endpoint/identity	Cleanup
Application copy	User-owned temporary files; immutable source copied, never executed in evidence	`/tmp/inside-orch-labs-cbfc94ade5b5/orch`	Remove only after evidence archived; preserve the canonical corpus
Go/cache/output	Compiler and downloaded module bytes; no system install	`/tmp/inside-orch-go`, `/tmp/inside-orch-gomod`, `/tmp/inside-orch-gocache`; binaries under run directory	Delete only exact run-owned paths, never global Go cache
API process	Loopback listener; disposable tokens and state	`ORCH_SERVER_ADDR=127.0.0.1:18090`	Send TERM to recorded PID and wait; do not kill by process name
Agent process	Docker socket privileged; set actual supported configuration	`ORCH_AGENT_ADDR=127.0.0.1:18091`, `ORCH_ADVERTISE_ADDRESS=http://127.0.0.1:18091`	Stop recorded PID before removing containers
Database	Destructive schemas allowed only in owned throwaway container	`inside_orch_lab_<run>`, PostgreSQL16; host binding `127.0.0.1:<allocated-port>:5432`	Remove exact labelled container and its exact dedicated volume, if created
Service containers	Docker create/start/remove; nginx image, dynamic host port	`atlas-api` in explicit lab namespace; record container IDs from API/Docker; orch labels alone insufficient ownership	Stop controller/agent, inspect each recorded ID, remove only exact owned IDs
Finite job container	Explicit once-only command, disposable alpine image	`atlas-count`; use JobSpec rather than the upstream redis service called worker	Remove exact recorded task/container IDs

Resource	Scope and privilege	Planned endpoint/identity	Cleanup
GitOps repo	User-owned temp Git repository, no push/credentials	`/tmp/inside-orch-<run>/gitops`, service identity remains atlas-api	Remove exact repo after recording commits and outputs
Storage fixture	Separate local directory/volume; no user paths	`/tmp/inside-orch-<run>/volumes`	Exact fixture only; no volume prune

Ports18090/18091 are suggestions and must be checked before binding; if occupied choose new loopback ports and record them. Workload published ports may bind all interfaces because the pinned Docker adapter does not expose a loopback HostIP field. Run those experiments in an isolated daemon/VM or accept that disposable workload exposure explicitly; do not claim `ORCH_SERVER_ADDR` restricts workload port binding. Dynamic allocation is node-scoped, not a cluster-wide reservation.

The upstream compose file publishes PostgreSQL5432, server8080, agent8081, demo18080, Traefik8088/8082/8443 and Prometheus9090 without loopback restriction. Do not launch it unchanged on a shared host. The Traefik profile uses `api.insecure=true`, and monitoring images include moving latest tags: these are upstream local examples, not a secure production prescription. Docker images for a measured run must have their resolved image IDs/digests recorded. No image or installer is fetched implicitly by the prerequisite checker.

Read-only capability check

From the website root:

```
python3 books/inside-orch/scripts/check_lab_prerequisites.py \
  --go /path/to/existing/go \
  --output books/inside-orch/validation/intake/lab-environment.json
```

Add `--probe-docker` only when a read-only Docker daemon version request is appropriate. The helper runs tool version commands with `GOTOOLCHAIN=local`, so invoking Go does not download a toolchain. It starts no process services, changes no infrastructure, and reports unavailable tools. The output file is an explicitly requested book evidence artifact.

Database reset guard

Provision a uniquely named disposable PostgreSQL container with labels `inside-orch.lab=true` and `inside-orch.run=<unique-run>`, `POSTGRES_DB=inside_orch_lab_<run>`, and only a `127.0.0.1` host binding for5432. Record its ID, image ID and binding. Keep its disposable password out of logs; provide the URL through `INSIDE_ORCH_LAB_DATABASE_URL` in the current process environment. Do not reuse any existing real database.

```
python3 books/inside-orch/scripts/run_database_tests.py \
  --container <recorded-container-id> \
  --working-copy /tmp/inside-orch-labs-cbfc94ade5b5/orch \
  --go /path/to/existing/go --check-only
```

Only after that passes, repeat without `--check-only`. The helper rejects remote hosts, wrong database names, missing ownership labels, mismatched published ports and altered source copies. It sets only `ORCH_INTEGRATION_DATABASE_URL` for the store suite. It deliberately does not enable E2E database tests, which exercise a different migration scope. The checker is a scope guard, not a guarantee against a malicious Docker daemon or concurrent administrator.

Interrupted-run recovery

Keep `run-resources.json` in the run evidence with exact PIDs, container IDs, volume names, namespace/service/task IDs and temp directories. Stop recorded server/agent PIDs first; verify PID command lines if recovering after interruption to avoid reused PIDs. Inspect each recorded container's labels/name/image and remove that ID only. Never run `docker`

`system prune`, `docker volume prune`, killall, or a broad `orch.managed=true` deletion. Remove only the explicitly named throwaway database. If ownership cannot be re-established, leave the resource and document the unresolved cleanup rather than guessing. Export observations before deleting state; redact bearer/bootstrap/registry credentials. Restart means a new process and new `MemoryService`, not reusing an object in a unit test.

Configuration anchors: `internal/config/config.go` server environment handling starts313; agent handling starts353. Default composition: `cmd/orch-server/main.go`142–158. Source documentation: `docs/DEVELOPMENT.md`, `docs/TESTING.md`; all references resolve under the pinned snapshot.

Observed baseline database fixture defect

The guarded run `validation/intake/postgres-store-owned.json` executed all three PostgresStore tests on the designated disposable database. `TestPostgresStoreWithTxRollsBackOnError` passed. The integration and concurrent-assignment tests failed with SQLSTATE42703 because node operations select `agent_version`, but their reset helper applies migrations000001–000014 and that column arrives in000016. This is an observed fixture/schema mismatch, not missing PostgreSQL tooling and not proof that assignment concurrency is broken. Preserve the failing raw log. The separately recorded all16-migration teaching fixture, validation/intake/full-schema-recovery-corrected.json, passed its guarded assignment and recovery checks; it is not silently substituted for the untouched upstream suite. The PostgreSQL E2E fixture applies only000001 and cannot establish later schema compatibility.

Practice questions

Keep answers separate while attempting the required artifact. Each rubric criterion is worth two points; partial answers must state their missing assumption. Source links refer to the pinned baseline.

DEV-01 — developer

Estimated time: 35 minutes. Prerequisites: Read Chapter 10 and the pinned source mechanism. Use only a disposable module copy; never mutate evidence. Go 1.25.0 and cached dependencies; Git installed for the GitOps fixture.

Two scheduler attempts hold the same pending task snapshot and expected UpdatedAt. Explain why one assignment wins, how the loser is represented, and whether exactly one runtime process is thereby guaranteed.

Required artifact: A three-step claim trace and the isolated test log.

Source: internal/scheduler/scheduler.go.

Scoring: one successful claim (2 points); one assignment event (2 points); physical fencing remains unproven (2 points).

Hint: Inspect persistAssignment and the fixture's success/event assertions.

Offline command in the disposable module:

```
go test -count=1 ./internal/scheduler -run
'^(TestConcurrentSchedulerAttemptsAssignTaskOnce)$'
```

DEV-02 — developer

Estimated time: 35 minutes. Prerequisites: Read Chapter 14 and the pinned source mechanism. Use only a disposable module copy; never mutate evidence. Go 1.25.0 and cached dependencies; Git installed for the GitOps fixture.

An assigned service task reports unhealthy while its policy permits replacement. Trace report status, stored actual state and replacement identity; distinguish a non-restartable service.

Required artifact: A status/identity table plus the named local test log.

Source: internal/controlplane/controlplane.go.

Scoring: incoming unhealthy retained in trace (2 points); stored failed for restartable branch (2 points); new identity instead of resurrection (2 points).

Hint: Find the policy conversion before task state storage.

Offline command in the disposable module:

```
go test -count=1 ./internal/controlplane -run
'^(TestUnhealthyRestartableTaskIsMarkedFailed)$'
```

DEV-03 — developer

Estimated time: 35 minutes. Prerequisites: Read Chapter 18 and the pinned source mechanism. Use only a disposable module copy; never mutate evidence. Go 1.25.0 and cached dependencies; Git installed for the GitOps fixture.

A task is actual failed. Compare a removed report under desired running and desired stopped, then show which evidence completes service deletion.

Required artifact: Two guard evaluations and a cleanup trace plus the test log.

Source: pkg/types/lifecycle.go.

Scoring: running-intent removal rejected (2 points); cleanup-intent removal accepted (2 points); all actual tasks removed required (2 points).

Hint: Read AgentCanReportTaskStatus in branch order.

Offline command in the disposable module:

```
go test -count=1 ./internal/controlplane -run
'^(TestAgentTaskStatusAllowsFailedTaskRemovalDuringDeletion)$'
```

DEV-04 — developer

Estimated time: 35 minutes. Prerequisites: Read Chapter 17 and the pinned source mechanism. Use only a disposable module copy; never mutate evidence. Go 1.25.0 and cached dependencies; Git installed for the GitOps fixture.

For desired replicas two, MaxSurge one and MaxUnavailable one, explain why a running new task may not permit old-task retirement or successful completion.

Required artifact: A version/status budget table and the rollout test log.

Source: internal/rollout/rollout.go.

Scoring: active budget 3 (2 points); minimum available 1 (2 points); new running is not newHealthy (2 points).

Hint: Compare newHealthy with the shared old availability helper.

Offline command in the disposable module:

```
go test -count=1 ./internal/rollout -run
'^(TestControllerSuccessfulRollout)$'
```

DEV-05 — developer

Estimated time: 35 minutes. Prerequisites: Read Chapter 27 and the pinned source mechanism. Use only a disposable module copy; never mutate evidence. Go 1.25.0 and cached dependencies; Git installed for the GitOps fixture.

Team A knows Team B's service UUID. Predict GetService under A's context, then distinguish this service test from a blanket claim about every workload endpoint.

Required artifact: A two-namespace lookup matrix and the namespace test log.

Source: internal/controlplane/controlplane.go.

Scoring: A list excludes B (2 points); B UUID yields not found for A (2 points); new workload paths require separate review (2 points).

Hint: Inspect the fixture's cross-namespace GetService assertion.

Offline command in the disposable module:

```
go test -count=1 ./internal/controlplane -run
'^(TestNamespaceIsolationAndDuplicateNames)$'
```

DEV-06 — developer

Estimated time: 35 minutes. Prerequisites: Read Chapter 28 and the pinned source mechanism. Use only a disposable module copy; never mutate evidence. Go 1.25.0 and cached dependencies; Git installed for the GitOps fixture.

Two one-replica services share MaxReplicas three. Concurrent requests each scale one service to two. Determine allowed final total and request outcomes without naming the winning service.

Required artifact: A race timeline and the quota test log.

Source: internal/controlplane/controlplane.go.

Scoring: one success (2 points); one quota denial (2 points); final total 3 independent of winner (2 points).

Hint: The fixture synchronizes goroutines with a shared start channel.

Offline command in the disposable module:

```
go test -count=1 ./internal/controlplane -run
'^(TestConcurrentScaleCannotExceedQuota)$'
```

DEV-07 — developer

Estimated time: 35 minutes. Prerequisites: Read Chapter 30 and the pinned source mechanism. Use only a disposable module copy; never mutate evidence. Go 1.25.0 and cached dependencies; Git installed for the GitOps fixture.

A local GitOps source manages api. Re-sync unchanged content, remove its manifest in a new local commit, and explain the prune boundary.

Required artifact: A revision/service-version trace and the local Git fixture test log.

Source: internal/gitops/gitops.go.

Scoring: unchanged revision no new version (2 points); prune limited to managed identity (2 points); model deletion differs from runtime cleanup (2 points).

Hint: Separate the previous managed set from the current manifest names.

Offline command in the disposable module:

```
go test -count=1 ./internal/gitops -run
'^(TestControllerSyncsLocalFixtureRepositoryAndPrunes)$'
```

DEV-08 — developer

Estimated time: 35 minutes. Prerequisites: Read Chapter 31 and the pinned source mechanism. Use only a disposable module copy; never mutate evidence. Go 1.25.0 and cached dependencies; Git installed for the GitOps fixture.

Using the fake-clock fixtures, compare Forbid and Replace with one active job, then explain catch-up with three missed minutes and MissedScheduleLimit two.

Required artifact: An occurrence/active-job table and both named test results.

Source: internal/cronjobs/cronjobs.go.

Scoring: Forbid no extra job (2 points); Replace changes identity (2 points); catch-up jobs 2 (2 points).

Hint: The cron controller has an explicit Clock seam; do not assume every package does.

Offline command in the disposable module:

```
go test -count=1 ./internal/cronjobs -run
'^(TestParseAndRunWithFakeClock|TestConcurrencyForbidAndReplace)$'
```

OPS-01 — devops

Estimated time: 45 minutes. Prerequisites: Read Chapter 24 and the pinned source mechanism. Use only a disposable module copy; never mutate evidence.

Ciphertext and its database backup survived, but the envelope key was lost. Design the recovery decision and secret-safe evidence collection.

Required artifact: A recovery checklist with key/ciphertext ownership and stop conditions.

Source: internal/secrets/envelope.go.

Scoring: matching key required (2 points); synthetic canary only (2 points); no automatic plaintext recovery claim (2 points).

Hint: Separate encryption metadata from recoverable key material.

OPS-02 — devops

Estimated time: 45 minutes. Prerequisites: Read Chapter 03 and the pinned source mechanism. Use only a disposable module copy; never mutate evidence.

A supervisor restarts the default server and /readyz becomes green. Services and node credentials disappeared. Explain the incident and bound the recovery action.

Required artifact: A before/after identity inventory and re-enrollment decision record.

Source: cmd/orch-server/main.go.

Scoring: MemoryService named (2 points); readiness independent of data (2 points); owned runtime reconciliation before replay (2 points).

Hint: Start at the constructor rather than the migration directory.

OPS-03 — devops

Estimated time: 45 minutes. Prerequisites: Read Chapter 39 and the pinned source mechanism. Use only a disposable module copy; never mutate evidence.

A fake-agent run reports high placement throughput. Prepare a report that keeps the result useful while rejecting a Docker or PostgreSQL performance claim.

Required artifact: A fully specified measurement sheet with run status and a claim boundary.

Source: cmd/orch-loadtest/main.go.

Scoring: full reproducibility inputs (2 points); fake-agent scope explicit (2 points); no Docker/database inference (2 points).

Hint: Read the runtime constructor used by the load tool.

OPS-04 — devops

Estimated time: 45 minutes. Prerequisites: Read Chapter 05 and the pinned source mechanism. Use only a disposable module copy; never mutate evidence.

The workload requests 250 millicores and 134217728 bytes. Docker inspection on the ordinary agent path reports both limits zero, while direct adapter construction applies them. Write a triage plan.

Required artifact: A field-flow diagram from service through assignment to ContainerSpec and engine inspection.

Source: internal/agent/agent.go.

Scoring: missing Resources field identified (2 points); request/reservation distinct (2 points); actual inspect boundary retained (2 points).

Hint: Compare the agent literal with the adapter's HostConfig mapping.

OPS-05 — devops

Estimated time: 45 minutes. Prerequisites: Read Chapter 23 and the pinned source mechanism. Use only a disposable module copy; never mutate evidence.

A heartbeat rotates a node credential, but its response is lost. The worker retries with the old token. Design a bounded diagnosis and enrollment recovery.

Required artifact: An authentication timeline with stored-hash/client-token divergence and safe stop condition.

Source: internal/api/api.go.

Scoring: rotation/receipt separate (2 points); no token logging (2 points); bounded authorized recovery (2 points).

Hint: Treat credential issuance and response receipt as separate events.

OPS-06 — devops

Estimated time: 45 minutes. Prerequisites: Read Chapter 19 and the pinned source mechanism. Use only a disposable module copy; never mutate evidence.

A node goes offline while hosting stateless Atlas API and stateful local-volume work. Explain replacement choices and how an old worker can still be running.

Required artifact: Two task/condition traces and a fencing risk decision.

Source: internal/controlplane/controlplane.go.

Scoring: stateless replacement trace (2 points); stateful locality retained (2 points); no physical fencing inference (2 points).

Hint: Keep Task conditions independent from actual/desired status.

OPS-07 — devops

Estimated time: 45 minutes. Prerequisites: Read Chapter 37 and the pinned source mechanism. Use only a disposable module copy; never mutate evidence.

All /healthz and /readyz requests succeed while work is pending and the database is unavailable. Build an observation plan that distinguishes these facts.

Required artifact: A three-layer health table with a verification probe for each layer.

Source: internal/api/api.go.

Scoring: process/workload/store separated (2 points); startup check not continuous (2 points); missing data remains unknown (2 points).

Hint: Read readyz, then find the actual server store constructor.

OPS-08 — devops

Estimated time: 45 minutes. Prerequisites: Read Chapter 35 and the pinned source mechanism. Use only a disposable module copy; never mutate evidence.

Design a future PostgreSQL restore rehearsal for an explicitly wired durable extension while preserving the default-server boundary.

Required artifact: A canary/digest recovery plan with key backup, schema, downtime and rollback criteria.

Source: internal/migrations/manager.go.

Scoring: all 16 schema and canary (2 points); envelope key retained (2 points); default memory limit explicit (2 points).

Hint: Restore completeness is an application-level property as well as a database command result.

ARC-01 — architect

Estimated time: 45 minutes. Prerequisites: Read Chapter 17 and the pinned source mechanism. Use only a disposable module copy; never mutate evidence.

For desired replicas 5, MaxSurge 2 and MaxUnavailable 1, compute the active and minimum-available budgets and identify which snapshots can violate a real availability promise.

Required artifact: A budget calculation and counterexample timeline.

Source: internal/rollout/rollout.go.

Scoring: max active 7 (2 points); min available 4 (2 points); stale observation counterexample (2 points).

Hint: Separate arithmetic from the freshness of its inputs.

ARC-02 — architect

Estimated time: 45 minutes. Prerequisites: Read Chapter 05 and the pinned source mechanism. Use only a disposable module copy; never mutate evidence.

Five replicas each request 250 millicores and 128 Mi. Compare their reservation with quota 1500 millicores and 700 Mi, then explain why passing arithmetic is not measured capacity.

Required artifact: A unit-correct table and an enforcement boundary statement.

Source: pkg/types/types.go.

Scoring: CPU 1250 (2 points); memory 671088640 (2 points); reservation not observed usage (2 points).

Hint: Use binary Mi=1048576 bytes and keep CPU in millicores.

ARC-03 — architect

Estimated time: 45 minutes. Prerequisites: Read Chapter 34 and the pinned source mechanism. Use only a disposable module copy; never mutate evidence.

Propose the minimum composition changes for durable intent and activated leader ownership, including a failure test that could falsify your design.

Required artifact: A labeled future-extension dependency graph and crash-test plan.

Source: cmd/orch-server/main.go.

Scoring: live owner replaced (2 points); leader loss reaches side effects (2 points); crash-window oracle (2 points).

Hint: List callers and ownership changes, not just new packages.

ARC-04 — architect

Estimated time: 45 minutes. Prerequisites: Read Chapter 34 and the pinned source mechanism. Use only a disposable module copy; never mutate evidence.

A canceled lease closes Done and Release returns nil, but another pool cannot acquire until the first pool closes. Explain the session-level failure and design a release test.

Required artifact: A two-pool lock ownership diagram and a corrected release contract.

Source: internal/leadership/postgres.go.

Scoring: original session ownership (2 points); Done not unlock proof (2 points); two independent pools (2 points).

Hint: Track database session identity across cancellation and deferred Release.

ARC-05 — architect

Estimated time: 45 minutes. Prerequisites: Read Chapter 26 and the pinned source mechanism. Use only a disposable module copy; never mutate evidence.

An HTTP probe's first destination is allowed, but it redirects to loopback metadata-like space. Define a safe synthetic test and the limitation of a URL-only validation check.

Required artifact: A resolver/redirect policy diagram with a zero-hit loopback oracle.

Source: internal/health/health.go.

Scoring: owned fixture only (2 points); redirect/dial policy evaluated (2 points); forbidden hits 0 (2 points).

Hint: The final connection target matters as much as the original URL.

ARC-06 — architect

Estimated time: 45 minutes. Prerequisites: Read Chapter 27 and the pinned source mechanism. Use only a disposable module copy; never mutate evidence.

A newer resource lookup takes only a global ID. Specify the minimum object-level namespace test even if user authentication already requires admin.

Required artifact: An authorization matrix for known/unknown/foreign IDs and a test-gap entry.

Source: internal/controlplane/workloads.go.

Scoring: foreign lookup matrix (2 points); mutation coverage separate (2 points); test gap not hidden (2 points).

Hint: Separate global role power from the method's declared tenancy contract.

ARC-07 — architect

Estimated time: 45 minutes. Prerequisites: Read Chapter 25 and the pinned source mechanism. Use only a disposable module copy; never mutate evidence.

An image string contains a syntactically valid sha256 digest. Explain which supply-chain claims remain unproved and propose additional evidence.

Required artifact: A syntax/resolution/content verification ladder.

Source: internal/image/reference.go.

Scoring: syntax support only (2 points); remote/content evidence distinct (2 points); future workflow labeled (2 points).

Hint: A reference parser does not perform remote registry resolution.

ARC-08 — architect

Estimated time: 45 minutes. Prerequisites: Read Chapter 38 and the pinned source mechanism. Use only a disposable module copy; never mutate evidence.

Define a task-convergence latency metric for an experiment with missing reports and restarts. Explain why an unconfigured alert is only a proposal.

Required artifact: A measurement contract with start/stop events, clock origin, censoring and scope.

Source: internal/metrics/prometheus.go.

Scoring: version-matched event boundaries (2 points); censored missing data (2 points); alert proposal explicit (2 points).

Hint: Define missing data before calculating a percentile.

Appendix D — worked solutions

These are reasoned answers to the 24 additional practice exercises, not fictional execution output. Named test logs are recorded separately.

DEV-01

Both attempts plan from the pending snapshot. The first compare-and-write changes NodeID and ActualStatus to assigned; the second stale claim conflicts. The scheduler treats that conflict as a benign non-assignment. The fixture requires exactly one successful assignment and one assignment event, not two. This proves a store/scheduler claim contract for one record. It does not fence a worker after a partition or prove default-server durability. Physical process uniqueness requires a separate execution/fencing design.

Full credit requires: one successful claim; one assignment event; physical fencing remains unproven.

DEV-02

The report enters as unhealthy; MemoryService checks service restart policy and converts it to failed when replacement is permitted. Failed is terminal for that task identity. Reconciliation creates distinct replacement work instead of making the same failed task healthy again. With replacement disallowed, unhealthy can remain retryable and later report running or healthy through valid transitions. Do not rewrite the agent's original report as failed or assume every unhealthy service automatically rolls back.

Full credit requires: incoming unhealthy retained in trace; stored failed for restartable branch; new identity instead of resurrection.

DEV-03

The generic transition helper permits failed to removed, but the agent guard rejects that change while desired remains running because terminal tasks accept only duplicates in that branch. Cleanup intent stopped or removed activates the cleanup report allowance, so removal is accepted. Service finalization still waits for all associated actual tasks to be removed; failed or stopped is insufficient. Label ownership must be checked by the agent before physical deletion.

Full credit requires: running-intent removal rejected; cleanup-intent removal accepted; all actual tasks removed required.

DEV-04

The active budget is three and the minimum available budget is one. Old availability can count running or healthy, while newHealthy counts only healthy target-version tasks. A running target task does not increment newHealthy. Repeated controller ticks therefore need explicit health transitions before the new population can satisfy the completion predicate. Desired stops and actual removals are separate: even a succeeded model state can precede physical cleanup.

Full credit requires: active budget 3; minimum available 1; new running is not newHealthy.

DEV-05

Same-name services may exist in separate namespaces with distinct IDs. ListServices under A returns A’s resources. GetService under A with B’s UUID must behave as not found, avoiding existence/metadata disclosure through the identifier. This specific tested service contract does not prove all newer job, volume or operation handlers perform the same lookup checks. Each path must validate namespace before returning a record. Authentication alone cannot replace object-level scope.

Full credit requires: A list excludes B; B UUID yields not found for A; new workload paths require separate review.

DEV-06

Starting total is two. Each accepted scale adds one, but the namespace maximum permits only one additional replica. Serialized mutation/accounting admits one request and denies the other with the quota error; final total is three. The winner is nondeterministic and must not be asserted by name. This memory-lock test is a local concurrency guarantee, not a transaction protocol for an unwired durable multi-writer deployment.

Full credit requires: one success; one quota denial; final total 3 independent of winner.

DEV-07

Initial sync records a revision and the managed api name. A repeated unchanged revision must not increment DeploymentVersion. Removing the manifest and committing a new local revision prunes the previously source-managed service; it does not authorize deleting every unrelated service in the namespace. The fixture creates a temporary local repository, uses no remote credentials and checks the deleted model state. Physical removal of running tasks remains asynchronous in a real agent deployment.

Full credit requires: unchanged revision no new version; prune limited to managed identity; model deletion differs from runtime cleanup.

DEV-08

Forbid leaves the existing active job and creates no overlapping job. Replace removes/replaces the active job identity in the fixture and creates one new active entry; Allow is a different policy. With creation at 12:00, now 12:03 and a per-minute schedule, the catch-up fixture admits only two jobs because the missed-schedule bound is two. It also records LastScheduleAt and NextScheduleAt. These are scheduling/model results, not proof arbitrary business effects execute exactly once across crashes.

Full credit requires: Forbid no extra job; Replace changes identity; catch-up jobs 2.

OPS-01

A ciphertext backup is unusable without the matching envelope key material. KeyID authenticates the envelope context but is not the decrypting key. Recover the exact key through an authorized independent recovery channel, prove decryption with a synthetic canary and never log real plaintext. If the key is irretrievable, record loss and re-provision secrets through application-specific recovery rather than claiming a database restore repaired them. Default MemoryService live secret records were never durable database state.

Full credit requires: matching key required; synthetic canary only; no automatic plaintext recovery claim.

OPS-02

The executable constructs MemoryService, so process restart loses its live maps. A successful listener restart and ready response do not restore desired services or issued credentials. Preserve sanitized pre-restart intent if available, reconcile any surviving owned containers carefully, then re-enroll disposable nodes and reapply reviewed intent in the laboratory. Do not query PostgreSQL as though it owned that lost state. Stop before adopting or deleting containers whose label ownership is uncertain.

Full credit requires: MemoryService named; readiness independent of data; owned runtime reconciliation before replay.

OPS-03

Record exact flags, seed, source SHA, Go/machine information, workload size, warm-up and duration with raw output. Explain that the tool drives in-memory control-plane behavior with fake agents. It does not pull images, start Docker containers or exercise the PostgreSQL store. Report

its denominators and error counts as actually emitted, without deriving unsupported startup latency or durable-cluster capacity. If no run exists, mark the sheet unexecuted and leave measured results absent.

Full credit requires: full reproducibility inputs; fake-agent scope explicit; no Docker/database inference.

OPS-04

Trace requested resources separately from runtime limits. The adapter can translate ResourceLimits into Docker Memory and NanoCPUs, but ordinary agent ContainerSpec construction omits that field. Therefore parser/DTO acceptance and adapter unit support do not establish the default end-to-end limit. Retain container IDs and inspect only captured disposable containers. Classify the missing propagation as an integration gap; do not silently patch source or promise enforcement based on quota reservations.

Full credit requires: missing Resources field identified; request/reservation distinct; actual inspect boundary retained.

OPS-05

After successful server rotation, the stored hash may correspond to the new token while the worker retains the old token. A retry can then fail authentication even though transport delivered the previous heartbeat. Record request IDs and expiry metadata without tokens. Confirm the exact node and laboratory enrollment authority, stop uncontrolled retry/re-enrollment loops, and use a deliberate recovery process. Neither token hashing nor a fifteen-minute TTL makes the response-loss window disappear.

Full credit requires: rotation/receipt separate; no token logging; bounded authorized recovery.

OPS-06

Stateless tasks receive node_lost metadata and are failed with desired removal so replacement can occur. Stateful tasks retain assignment for manual recovery because node-local data is not replicated or moved by a status update. A partitioned worker can still execute either old process. Offline detection is suspicion based on heartbeat age, not physical shutdown. Do not force stateful failover without a storage and fencing plan; stop if ownership or old-writer quiescence cannot be established.

Full credit requires: stateless replacement trace; stateful locality retained; no physical fencing inference.

OPS-07

Process liveness/readiness confirms the HTTP process's reported state; the handlers do not prove a database-backed live control plane or workload health. Inspect task assignments, actual/container health and specific operation errors separately. If schema check is enabled, database access can gate startup, but a past startup check is not a continuous durability guarantee. Proposed alerts need configured collectors and tested thresholds. Keep missing workload samples unknown rather than healthy.

Full credit requires: process/workload/store separated; startup check not continuous; missing data remains unknown.

OPS-08

Use an owned disposable database, all sixteen migrations, known canary records and an exact dump/restore procedure. Retain the envelope key independently and compare restored record identities and content digests, not just process exit codes. Record restore duration and tolerated data-loss window from defined event origins. Do not claim the default memory server's live intent is in this database. Stop on schema mismatch, missing keys or canary divergence; never overwrite a production database as an exercise.

Full credit requires: all 16 schema and canary; envelope key retained; default memory limit explicit.

ARC-01

The model allows up to 7 active tasks and requires at least 4 available by its accounting. New target-version running tasks still do not count toward newHealthy. An old task can fail physically after the snapshot but before a stop is applied, making the prior budget stale. Client reachability can fail despite local health. These counters guide controller decisions; they do not establish a continuously enforced service-level objective.

Full credit requires: max active 7; min available 4; stale observation counterexample.

ARC-02

CPU reservation is 1250 millicores. Memory is 5×134217728=671088640 bytes, equivalent to 640 Mi. Both fit the stated 1500-millicore and 734003200-byte quotas before considering any other services. These are requested reservations, not observed process usage or proof Docker limits are applied. Include existing namespace consumption in a real quota decision and inspect field propagation before promising enforcement.

Full credit requires: CPU 1250; memory 671088640; reservation not observed usage.

ARC-03

Replace the live state-owner construction with a complete durable service implementation rather than merely opening a pool. Wire controllers to durable transactions, restore identity/operation state and add active leader-loss cancellation tied to side-effect authority. Specify schema, credential and envelope-key recovery. A falsifying test kills the

process after durable intent but before runtime/report completion and checks reconstruction plus duplicate suppression. This is future architecture; the current binary does not supply these guarantees.

Full credit requires: live owner replaced; leader loss reaches side effects; crash-window oracle.

ARC-04

A session-level advisory lock belongs to its original database connection. Cancellation can release a pooled connection before unlock, allowing a later unlock call to run on another session and falsely appear sufficient. Done closure signals local ownership loss but does not prove remote lock release. Test two independent pools: acquire A, cancel/release A, attempt B before and after closing A's pool. Repair requires unlocking on the original pinned connection or closing that session; default server HA remains unwired.

Full credit requires: original session ownership; Done not unlock proof; two independent pools.

ARC-05

Use only owned loopback fixture servers with synthetic responses and controlled resolution; do not contact live metadata services. Validate destination policy for every connection/redirect path, accounting for DNS rebinding and address changes rather than checking only the initial string. The forbidden listener should observe zero hits. Configuration validation and generic HTTP client behavior are separate boundaries. Retain exact resolver and redirect behavior from the pinned health implementation before claiming SSRF prevention.

Full credit requires: owned fixture only; redirect/dial policy evaluated; forbidden hits 0.

ARC-06

Create two namespaces and distinct resources, then request each foreign ID under the other namespace context. The result must avoid returning foreign metadata and should match the intended not-found policy rather than disclose existence. Test list, get, delete and mutation paths separately. Admin fallback authentication does not by itself establish namespace-scoped lookup or delete semantics. If the current method lacks a scope check, report the gap instead of extending older service tests to it.

Full credit requires: foreign lookup matrix; mutation coverage separate; test gap not hidden.

ARC-07

The parser can validate digest form and admission policy can require a digest. That does not prove a remote registry returned the intended manifest, that all referenced layers were verified, or that a tag's current mapping equals the recorded digest. Propose an explicitly authenticated registry resolution and content-verification workflow with immutable returned digest and provenance records. Keep this future workflow separate from baseline syntax support and do not invent adoption or integrity results.

Full credit requires: syntax support only; remote/content evidence distinct; future workflow labeled.

ARC-08

Start at accepted intent version timestamp and stop at the first authoritative matching task health/required replica observation. Preserve one monotonic process clock for local elapsed durations or document wall-clock skew across processes. Mark missing completion as censored/timeout rather than zero latency, and separate restart-lost records from

successful convergence. Specify workload size, warm-up and sample denominator. A dashboard expression and suggested threshold do not become an operational alert until configured and tested.

Full credit requires: version-matched event boundaries; censored missing data; alert proposal explicit.

Chapter 1 - Worked answers

1. Acceptance and execution

CreateService stores desired service state and may create assigned tasks, but it does not call Docker. The API response establishes accepted intention in one MemoryService. The agent later polls and calls its runtime. A healthy replica requires an accepted report and, for operational evidence, an independent Docker/endpoint observation. A task ID correlates these facts without making them simultaneous. See controlplane.go1253–1323 and agent.go253–265 at the chapter's pinned links.

2. The state owner

Server main146 calls controlplane.NewMemoryService. Its constructor allocates maps, including services and tasks. The optional ORCH_SCHEMA_CHECK branch runs before this constructor and opens a migration runner only to read schema status. A successful migration changes PostgreSQL tables, not the constructor or backing store of those maps. Choosing durable storage is a composition/service-layer change, not a database URL setting.

3. The unsafe readiness assumption

The ready handler returns a fixed success response. Still missing are the pre-restart service/task IDs in the new API and the actual state of containers left on workers. Also check node credential continuity and the endpoint independently. If the API is empty while Docker has running containers, report that split observation; do not call the old workload recovered or safely removed. A controller restart using the same service instance would not test this failure.

4. Deployment choice

The disposable training cluster fits the baseline when its state may be discarded and the owner can inspect/clean all resources. The payroll workload's stated requirement is not met: fresh server maps lose desired state, and a database package is not integrated. A justified next step is to evaluate an established platform or design/test a separate durable service composition with feature parity and process recovery. Neither choice requires pretending the existing code is useless; its inspectable seams are valuable precisely because their limits are explicit.

These are reasoning answers checked against source. They do not claim an unrecorded live deployment or process-restart observation.

Return to Chapter 1.

Chapter 2 - Worked answers

1. Follow both implementation and composition

Runtime is defined in internal/docker/docker.go47–56. EngineRuntime implements daemon calls in that file; FakeRuntime is in internal/docker/fake.go and records operations. cmd/orch-agent/main.go74–83 constructs and wraps the engine adapter, then injects it into Runner. The declaration describes a contract, the implementations describe two possible worlds, and the constructor chooses the world used by this binary. An intended internal/runtime directory would prove none of these facts.

2. Planner fixture scope

The fixture checks deterministic assignments for explicit nodes, services, pending tasks and existing resource use. It can assert no assignment on insufficient resources, bad labels or offline/draining nodes. It cannot establish that normal CreateService invoked Plan. MemoryService.reconcileServiceTasksLocked creates assigned tasks directly; compare that caller independently before claiming all ordinary deployments have planner filtering.

3. Controller restart versus process restart

A new controller using an old MemoryService tests resumption against preserved maps. It may prove repeated reconciliation avoids duplicate work for those fixture inputs. A new process creates a new MemoryService and loses those maps. The unsafe claim is process/database durability. To test it, use distinct processes and compare persisted IDs before and after, not a reused pointer.

4. Honest tool statement

“make lint completed its available fallback: tracked-Go gofmt check and go vet. golangci-lint was unavailable and was not executed.” The remaining check is the configured golangci-lint version and rules. Also record the source export/VCS context because the fallback uses git ls-files and a source copy without Git history needs deliberate handling. Do not label the fallback as a full lint run.

Reasoning answers were checked against pinned source. The chapter's selected Plan and fake-runtime tests were executed with the exact recorded command; no answer claims real daemon/database evidence.

Return to Chapter 2.

Chapter 3 - Worked answers

1. Startup has several boundaries

The script waits only on readyz, whose fixed success response means the handler is reachable. Agent registration is later/asynchronous, so node ls must show the owned ready node. dev-up starts postgres/server/agent but contains no migration invocation. The separate migrate-up script applies SQL; neither script changes the server's NewMemoryService selection.

2. Host ports are finite per node

Public zero delegates dynamic allocation; each task can receive a distinct host port while both container endpoints use port80. A fixed published port/protocol cannot be bound twice on the same host address. Planner filtering, memory helper behavior and actual Docker collision are distinct paths. Scaling desired replicas does not create a second physical port capacity slot. Inspect assigned task ports and Docker bindings; do not assume resources/labels were checked by Plan on normal service creation.

3. Keep the cleanup executor available

The worker owns the runtime stop/remove actions. Stopping it first can leave containers on the host daemon even after Compose removes the worker service. First ask the still-running server to delete, wait for reports, then inspect exact recorded container IDs. If the server/agent is already gone, use those identities to assess and remove only the disposable experiment's objects. A broad orch.managed filter is insufficient ownership proof on a shared daemon. Preserve PostgreSQL volumes unless intentionally deleting owned disposable data.

4. The constructor is decisive

Database availability is not state ownership. The server constructs MemoryService even with DatabaseURL set. A narrow test creates a named namespace/service through one owned loopback server, saves IDs, terminates only that server, starts a new server with the same configuration, and compares API state. Correlate surviving Docker objects separately if a worker is involved. The authenticated fresh-process record demonstrates memory loss without making this a database-recovery claim; the full Compose walkthrough remains separately scoped.

The upstream commands were checked against source. A source walkthrough and compiled CLI do not by themselves certify the live Compose sequence.

Return to Chapter 3.

Chapter 4 - Worked Answers

1. Correlate a replica

Keep task `id`, `service_id`, `version`, `node_id`, `container_id`, `desired_status`, and `actual_status`, together with namespace and observation time. The container ID can be absent before creation. Service name identifies intent for a human but cannot distinguish two replicas, an old deployment version, or an identity created after deletion. Compare the report's task identity against the actual assigned task before updating any state.

2. Separate generic legality from report acceptance

The actual-state validator allows unhealthy to healthy, rejects failed to running, and allows failed to removed as cleanup. The agent gate adds conditions. With desired running and actual failed, only a duplicate failed report is accepted by the terminal-state branch; a removed report is rejected. With desired removed, cleanup reports including removed are accepted by the earlier desired-state branch. The order of checks matters. The generic edge does not by itself authorize an agent report. These conclusions follow directly from lifecycle.go136–187 and302–318 and the selected executed lifecycle tests.

3. Interpret three different objects

Service active describes eligibility for reconciliation. Deployment succeeded records the terminal result of a particular rollout operation. Task failed describes a particular execution identity. A later container failure does not rewrite the historical rollout result into a new result. A policy-driven replacement creates another task, associated with the service's intended version. Whether that replacement is durably recorded or recovered after restart requires inspecting the selected control plane and store, rather than reading the type definitions.

4. Reject stale health after cleanup intent

A later arrival time does not mean a report reflects later server intent. The agent could have observed health before receiving the scale-down instruction. AgentCanReportTaskStatus rejects healthy when desired status is removed, allowing only cleanup reports. Keep the task and report identifiers and inspect the desired-state branch. Do not turn an event timestamp into a distributed total order. This rejection preserves deletion intent while the next assignment poll teaches the agent what cleanup is required.

Return to Chapter 4.

Chapter 5 - Worked Answers

1. Keep units throughout

0.4 CPU becomes400 millicores. Five replicas request2,000 millicores, or2 CPU. 64Mi is67,108,864 bytes per replica, totaling335,544,320 bytes, or320Mi. 64M is64,000,000 bytes per replica and320,000,000 bytes for five. The decimal/binary difference is15,544,320 bytes in the aggregate. These totals describe requested budget, not measured consumption or a proven host limit.

2. Identify the rejecting stage

The CPU case reaches ParseCPU and rejects an unparseable positive quantity. The memory case reaches ParseMemory, whose Mi branch requires a positive integer, so1.5Mi is invalid. The unknown replica field is rejected earlier by the YAML decoder's KnownFields mode. The actual fixture harness record includes the separate child commands and diagnostics, rather than treating a nonzero exit as sufficient evidence of any particular validation rule. No server is contacted by validate.

3. Compare actual paths

A node cannot provide the same host TCP8088 binding to two ordinary containers. The scheduler planner reserves node/protocol/port keys and rejects a second candidate. Immediate MemoryService task creation uses another helper which reserves but does not reject explicit conflicts. Therefore scheduler correctness alone does not protect this service path. The safe lab does not create a real shared-host collision; the source-confirmed gap is enough to reject the assumption. Dynamic public zero changes allocation behavior but cannot detect unrelated host processes outside orch's known tasks.

4. Trace caller data

Service resources are normalized and can be counted, but AgentTask does not transport them and ensureTask does not set ContainerSpec.Resources. The adapter's HostConfig code cannot enforce values it never receives. The real Docker contrast retained in the intake record shows zeros through the ordinary service path and the expected bytes/NanoCPUs through direct adapter invocation. To support isolation for Atlas itself, repeat an owned end-to-end service-path inspection after an independently reviewed implementation change. Passing a direct adapter test or modifying the book's baseline would not supply that evidence.

Return to Chapter 5.

Chapter 6 - Worked Answers

1. Name active owners

GitOps, cron, scheduler, rollout, and node monitor are all constructed and run around the same memory object. Usage snapshots and retention pruning add separate periodic activities; DNS is conditional. Immediate creation is different: CreateService calls reconcileServiceTasksLocked inside its mutation critical section and produces assigned tasks before a periodic scheduler tick. The existence of that ticker cannot prove every creation used Plan's checks.

2. Explain local atomicity

Image resolution, initial validation, admission, normalization, and revalidation occur before locking. Namespace lookup, duplicate-name detection, secret/registry existence, quota checking, service map creation, version storage, immediate task reconciliation, and event append occur under the mutex. This coordinates concurrent users of this MemoryService instance. It does not create a durable database transaction or include a Docker operation in the critical section. A process exit can discard all those completed map changes.

3. Bound the restart claim

The harness created the container directly, then authenticated a status report containing its exact ID for the assigned task. That report established a server-visible association before restart. No upstream Runner created or adopted the container. The second binary lost its task maps while Docker retained the same object. To claim automatic adoption, a separate real-agent experiment would need to show how a returning agent identifies surviving containers, reconnects their task identities, and handles missing server intent safely. The current record does not contain that behavior.

4. Reject configuration-only durability

The active constructor creates MemoryService regardless of DATABASE_URL. Optional schema checking and migration commands are separate database operations. The measured second process recreated only default namespace and had no prior services, tasks, nodes, or custom namespace. Recovery therefore needs a durable selected implementation plus complete reads/writes and reconstruction, not simply a URL. Restored tables cannot populate maps through a code path that never reads them. Avoid recreating competing runtime work while surviving containers remain unexplained.

Return to Chapter 6.

Chapter 7 - Worked Answers

1. Read outer wrappers first

requestIDMiddleware and apiVersionMiddleware wrap the authentication middleware. They set response metadata before the user-auth check returns a rejection, so configured JWT failures can still be correlated and versioned. The header does not prove authentication succeeded or that a domain mutation occurred. Namespace validation also precedes role checks within user-auth middleware.

2. Handle a malformed identifier

Branch on status400 and code invalid_argument, as verified by the public-client smoke test. Display or retain the message as human guidance, and retain RequestID for log correlation. Do not match the complete message text or interpret RequestID as a commit record. A404 would describe another condition, while a transport failure may not contain any API envelope. The client preserves status even when an intermediary returns an unexpected error body.

3. Distinguish addition from rename

The fixture adds future_hint while retaining service.spec.name, so the ordinary JSON decoder ignores the unfamiliar field and preserves the known name. Renaming name to service_name removes the field the old client knows. Its decoder ignores the new field and leaves Name empty, even though syntax decoding succeeds. This is a semantic compatibility failure demonstrated by a controlled fixture, not a change to the real API. Both fixtures were executed in the recorded smoke test.

4. Respect the intended audience

The upstream test explicitly forbids /v1/agent/ paths in the public schema. Those use node-registration and agent-report contracts and should not be counted as public omissions. The genuine compared

omissions are GET /v1/discovery/services, GET /v1/discovery/services/{name}, and GET /v1/integrations/traefik/config. DOC015 preserves their exact router anchors. Static method coverage does not establish correctness of all schemas or configured authentication; those remain additional checks.

Return to Chapter 7.

Chapter 8 - Worked Answers

1. Predict nonempty precedence

Nonempty flags win, followed by nonempty environment values, then YAML. Removing flags allows environment to win; removing that environment setting exposes the YAML value. An empty override does not erase a lower-level nonempty value. Namespace ultimately defaults to default, while server URL remains required. The actual CLI lab exercised server and namespace conflicts. Token follows the same source-inspected loader order, but the lab intentionally does not claim a multi-token authentication experiment.

2. Distinguish local and network work

Validate invokes ParseDeployFile and returns parser/validator outcomes; it never calls the client factory. Service ls constructs a client, which first resolves a configured server URL and fails with actionable guidance if absent. A valid file can still be rejected by a server's policy, authorization, namespace or quota. Local validity therefore cannot be used as a deployment-success result.

3. Check the current version

The checker uses deployment_version from the bare service object, task version, desired_status and actual_status from each task, and spec.replicas. An old-version healthy task is not counted toward the current version. Current-version assigned tasks are not reported available, so the script exits one. This is a legitimate convergence failure rather than a JSON error. The two reads are not an atomic snapshot; a concurrent rollout can justify retrying the observation with a deadline, not changing intent automatically.

4. Preserve specific failures

A table grep can match an old replica, one of several replicas, a desired-running column, or diagnostic text. It also loses structured identity and version. The teaching checker validates the JSON shape, requires all current intended replicas available, and separates non-convergence from command/transport failure and incompatible data. The actual lab observed one before node registration, zero after manually accepted running reports, and two for missing service and connection refusal. It proves server-observed convergence only; no Docker or fresh endpoint health was measured.

Return to Chapter 8.

Chapter 9 - Worked Answers

1. A stale task claim

AssignTask predicates on task ID, expected UpdatedAt and actual_status pending. After the first writer updates that row, the second writer holding the old token cannot match the same pending snapshot. A subsequent scan with no returned row becomes ErrConflict. Although row_version increments, the shown WHERE clause does not compare it, so its presence alone cannot explain the concurrency guard. Read the actual winner as well: two rejected requests would not establish progress. The separate full-schema experiment observed one winner and one conflict; the untouched upstream fixture failed earlier because its migration list was incomplete.

2. Escaping the transaction

The callback must use its transaction-scoped store argument. A closure can still call a captured outer pool-backed store; lexical nesting does not change which connection executes SQL. That outer call can commit independently even if the callback later returns an error. The generic WithTx fallback also executes directly when its candidate does not implement Transactor. Therefore identify the concrete candidate and trace every relevant mutation and strict event through the scoped argument. A Docker call cannot be rolled back by returning a PostgreSQL error; external effects require reconciliation or an explicitly designed protocol.

3. A fixture/schema mismatch

The observed SQLSTATE42703 failures mean current node SQL selected agent_version from a schema without that column. The upstream reset helper applies migrations 000001–000014; agent_version is added by 000016_operations_safety.up.sql. Preserve those failures as baseline evidence. In a separately named teaching fixture, apply all 16 migrations to an owned disposable database, create one pending task, give two

writers its same UpdatedAt, and assert one success, one conflict and a coherent stored assignment. That exact separate experiment passed in the recorded full-schema fixture; it does not rewrite the upstream test result.

4. Durable composition evidence

Inventory every API state and ownership relationship, including jobs or notification sinks, whose interfaces are not embedded in the composite Store. Persist intent, identities, credentials/secret references and operation history with defined transaction boundaries. Start an application instance, record Atlas service/task identity, terminate it, construct a new process against the same database, and verify restored API state and subsequent reconciliation. Reusing one fake store or MemoryService object only tests controller reconstruction. Also test schema compatibility, failed side-effect recovery and secret-key requirements. A constructor change alone neither supplies missing methods nor proves safe restart behavior.

These answers distinguish source reasoning from the scoped execution records linked in chapter 9. Proposed changes require separate implementation and validation.

Chapter 10 - Worked Answers

1. Eligibility precedes scoring

Node b already reserves fixed TCP port 31080. Atlas requests that same published port, so b is rejected before its larger remaining memory can win a score comparison. Node a is offline. Node c is ready and has sufficient CPU, memory and the requested port for atlas-01. After that assignment, c still has 600m and 128Mi available, but its fixed port is reserved. The second replica therefore remains unassigned. The executed teaching fixture asserts this result and permutation invariance without running containers.

2. Dynamic published ports

With the original unrelated reservation still present on b, a dynamic Atlas port can avoid 31080. Both b and c can now be eligible. For the first replica, b has more memory remaining after placement and wins that first score component. After reserving 128Mi there, b has 320Mi before the second placement, versus c with 256Mi; its post-placement remaining memory is still larger, so the second replica also prefers b when a distinct dynamic port is available. Do not assume dynamic ports imply spreading. The same-service and total-task tie-breakers only matter after remaining-memory equality. This is a reasoning answer; the chapter recorded the fixed-port fixture, not this modification.

3. Distinct rows do not serialize capacity

Each writer can successfully assign a different pending task using that task row's valid optimistic token. Both may have read the same node resource budget and treated it as free. Winning both task claims does not atomically subtract from a shared reservation row. The Plan-local clusterState prevents conflicting selections within one invocation, but separate invocations do not share that temporary object. To claim global

reservation correctness, test an atomic shared reservation or a deliberately serialized/revalidated allocation protocol under concurrent planners, including fixed-port collisions.

4. Two missing integration edges

First, ordinary MemoryService service creation makes assigned tasks directly, so Plan's property assertions cannot prove every live deployment passed through its resource filters. Second, the agent's ContainerSpec literal forwards command and volumes but omits Resources, even though the lower Engine adapter can translate resource fields into HostConfig. Planner accounting is not kernel enforcement. Trace both the placement path and the complete resource field path, then inspect the actual container's limits under an owned daemon before making the operational claim. A passing pure function property test establishes neither edge.

These answers distinguish source reasoning from the scoped execution records linked in chapter 10. Proposed changes require separate implementation and validation.

Chapter 11 - Worked Answers

1. Running plus pulling

The next pass creates zero tasks. Both current-version tasks are nonterminal and enter current, so effectiveCount is two. Health is not the criterion for this replica deficit calculation. Counting only healthy tasks would recreate work while image pull or initialization was still in flight, potentially growing replicas every iteration. Availability and rollout readiness require their own observations and policy; they should not be silently substituted into this count.

2. A failed task and restart policy

With restart never, the failed current-version task contributes to nonRestartableFailed. One running plus one such failed task produces effectiveCount two and no replacement. With on_failure, the failed task does not enter that special count, so effectiveCount is one and one replacement is created. The policy prevents endless replacement of deliberately non-restartable work. This library count alone is not evidence of retry backoff, a maximum restart budget or execution of the replacement container.

3. What the restart fixture retains

The fixture constructs a new reconciler while retaining the fake store that contains the first controller's tasks. The new controller can read those same identities and avoid duplicate creation. Deleting or replacing the fake store removes the authoritative input and tests a different scenario. Neither version establishes fresh-process recovery of the default server's maps. The focused executed suite supports the retained-store controller behavior; database composition and actual host containers remain separate boundaries.

4. Two controllers with the same deficit

Use a barrier fake store so two reconciler instances both finish ListTasksByService with an empty list for desired two before either creates a task. Release both and inspect all created identities and events. Per-task optimistic guards cannot protect creation of previously nonexistent IDs. A globally unique creation claim requires a serialization mechanism covering the deficit read and creates, such as a correctly fenced leader or an appropriate shared transaction/locking protocol. Default NoopLeaderLock is not that mechanism. Define the protocol and test stale ownership before asserting global uniqueness.

These answers distinguish source reasoning from the scoped execution records linked in chapter 11. Proposed changes require separate implementation and validation.

Chapter 12 - Worked Answers

1. Invocation versus object identity

The fake operation recorder can log two CreateContainer calls while the implementation returns one existing container identity. Idempotency concerns the resulting managed object set, not the number of method invocations. Assert that both results identify the same task-owned container and that the managed container set has not grown. A matching image name alone is insufficient ownership evidence. At the Engine adapter boundary, task-ID lookup and create-conflict recovery are narrower than a complete desired-spec comparison.

2. Uncertain inspection

Propagate or record the daemon transport error and avoid treating it as proof of absence. Creating or removing based on an unavailable inspection can duplicate work or damage a still-running container. Not-found has explicit semantics; a disconnected socket or server error does not. A safe fixture should inject these outcomes separately and assert no destructive operation for the uncertain case. A later successful observation can drive convergence without rewriting the original failure as a clean absence.

3. An unrelated recorded ID

The task record is a hint about runtime identity, not authority to delete any referenced object. The agent inspects ownership metadata and requires the managed marker plus matching node/task identity before removal. A nonmanaged container fails that predicate even if the task's ContainerID points at it. The focused tests passed the nonmanaged protection scenario using fakes. They do not authorize deleting arbitrary containers on a real host; an owned-daemon repetition must inventory exact run resources and preserve unrelated objects.

4. Lost acknowledgement adoption

On an isolated daemon, deploy one Atlas task, arrange for its created report response to be lost after server acceptance, and retain the real task/container IDs and managed labels. Reconcile again and verify that the same suitable container is adopted rather than another object being created. Record actual reports, Engine object count and IDs before cleanup. This demonstrates recovery across that specific acknowledgement window. It does not establish default-server durability: a new server process loses its memory-owned task inventory unless a separate durable composition restores it.

These answers distinguish source reasoning from the scoped execution records linked in chapter 12. Proposed changes require separate implementation and validation.

Chapter 13 - Worked Answers

1. Retry scope

Registration and heartbeat use the generic retry helper. reportTaskStatus makes one client call; it does not invoke that helper. Recovery can still occur through a later reconciliation pass: an already-running suitable container can be observed and its running state reported again. This depends on retained identities and ownership metadata, not on pretending the first report succeeded. A passing re-report test is narrower than a retry guarantee for every operation or every permanent authentication error.

2. Accepted report with a lost response

The server may already hold running while the agent sees a transport error. The next pass uses the same node ID, task ID and suitable managed container identity to observe the existing process and report again. Server lifecycle and ownership checks decide whether the duplicate is acceptable. Preserve those identities in the experiment and assert no extra container was created. This is protocol recovery while the server retains its state; the default MemoryService does not make those identities durable across a new server process.

3. Logical guards versus physical fencing

Rejecting an old running report prevents the accepted control-plane task record from moving backward or being resurrected after removal. Marking a node offline permits a policy response to missed heartbeats. Neither action proves the partitioned node's process stopped or lost access to external resources. Physical fencing requires a mechanism that actually revokes execution or resource authority and an experiment showing stale ownership cannot continue its side effects. Do not promote a state-machine test into that stronger distributed-system guarantee.

4. Cancellation has multiple observables

Cancel Runner.Run while a fake client request and runtime log operation are deliberately blocked. Assert the runner returns, the client request context is canceled, runtime channels close and owned listener/connection resources stop. Include time-bounded failure assertions so a leaked activity is visible. Separately test shutdown notification's fresh bounded context, because an already-canceled run context cannot deliver it. A cooperative fake demonstrates the injected contract; testing a real Engine blocked reader requires its own bounded local experiment and cannot be inferred solely from runner return.

These answers distinguish source reasoning from the scoped execution records linked in chapter 13. Proposed changes require separate implementation and validation.

Chapter 14 - Worked Answers

1. Consecutive thresholds

Set both healthy and unhealthy thresholds to two and start from running. The sequence 200,404,200,404,404 produces counters successes/failures of 1/0,0/1,1/0,0/1,0/2. No healthy report is justified because two successes never occur consecutively. The last observation reaches two failures and triggers unhealthy. Total failure count three is irrelevant to this decision. This reasoning assumes ordinary results, not checker errors, whose early-return branch does not update these threshold counters.

2. TCP versus HTTP semantics

A TCP probe can connect successfully while Atlas returns503 on its readiness route, sends invalid HTTP or never produces a useful application response. TCP checks connection establishment, whereas the HTTP checker also receives and classifies an HTTP status under its timeout and redirect policy. Choose the probe for the intended observation. Neither one independently establishes every business dependency or complete traffic-path availability.

3. Origin includes port

A same-scheme redirect to another path on the same host and port passes the same-origin comparison, subject to the redirect-count bound. A redirect from one loopback port to another changes URL.Host and is rejected even though both addresses are 127.0.0.1. sameOriginRedirect compares lowercased scheme and host against the original URL. The executed loopback suite exercised these redirect cases; it did not establish a universal network allowlist for all callers.

4. The caller establishes the initial target

DefaultChecker accepts a Check.Target. Its redirect restriction keeps that request on the supplied origin but does not independently restrict every possible initial address. The agent's validated manifest/assigned-port path generates the loopback target. Calling the checker directly bypasses that caller boundary. A safe test uses two owned loopback servers: invoke the checker directly with one supplied initial URL, then show the agent builder rejects an unassigned port. Keep both servers local, record exact requests and clean up their listeners; there is no need to probe private or external addresses.

These answers distinguish source reasoning from the scoped execution records linked in chapter 14. Proposed changes require separate implementation and validation.

Chapter 15 - Worked Answers

1. Generic events and contextual meaning

The actual fixture returns four task.status events, but their generic message does not encode pulling, created, running and healthy separately. The fixture's real request sequence supplies those submitted values, and accepted task snapshots provide additional state context. The event count alone therefore cannot prove four particular lifecycle transitions or any Docker execution. The fixture explicitly injects its reports and uses a no-Docker container identifier.

2. Best effort, strict and transactional

Best-effort Emit logs an append error and returns nil. Strict Emit propagates that error to its caller. Only a caller that joins its mutation and strict append in the same effective database transaction can use the error to roll both SQL effects back. Strict append after an independently committed mutation cannot undo the earlier commit. The API's best-effort audit path is another distinct policy; a successful user response does not imply every audit append succeeded.

3. Duplicate external acceptance

The receiver can accept the first POST while its response is lost. The sender retries the same event payload and the receiver may accept it again. Deduplicate using stable event identity, scoped by the producing system/namespace as needed, and record completed receiver actions atomically with the deduplication decision. An HMAC signature authenticates payload bytes relative to the shared key; it neither encrypts them nor prevents duplicate valid deliveries. The executed fake transport checks retry attempts and selected redaction/signature behavior, not an actual receiver's deduplication implementation.

4. An empty list after restart

Possible causes include no matching events, the wrong namespace/type/service filter, retention, or loss of MemoryService state in a fresh process. Compare pre-restart captured IDs/timestamps and filters, process identity, retention settings and separately retained audit/log evidence. Empty history alone cannot prove no operations occurred. PostgreSQL event tables do not change the default executable's storage path. A durable history claim requires actual composition and a fresh-process recovery experiment, not only an implemented store package.

These answers distinguish source reasoning from the scoped execution records linked in chapter 15. Proposed changes require separate implementation and validation.

Chapter 16 - Worked Answers

1. Information lost in plain text

LogLine carries stdout/stderr identity and a read-time timestamp. The final handler prints the text line, so those structured fields disappear. Concurrent scanners can interleave outputs, and visible ordering cannot reconstruct which source stream produced identical text. A timestamp assigned at scanning is also not necessarily the application event's original time. Preserving that information would require a separately documented structured public response format and compatible clients.

2. Tail versus follow

Tail selects how much existing history the Engine should return; follow asks to continue as new output arrives. They are independent. In the worker handler, a malformed tail string falls back to all. A negative string that parses as an integer follows a different path and is passed onward; do not describe all invalid-looking inputs as identical. This answer traces current parsing, not a recommendation to request unbounded logs. Use a small explicit tail and a bounded client deadline in the live lab.

3. Cooperative cancellation scope

The passing fake log fixture waits on context cancellation and then closes its channels. That establishes the handler/fake contract. Real Engine streaming includes a response reader, stdcopy, two pipes and scanners that can block before reaching the context-aware send. A select around the output send alone cannot prove every blocked read wakes on disconnect. A separate real local experiment must inspect reader/pipe closure and bounded goroutine completion under its own failure schedule.

4. A bounded slow-consumer extension

Define a maximum line/byte queue, a client write deadline and an explicit overflow policy such as closing the stream with a recorded diagnostic. Avoid a silent unlimited buffer. Inject a consumer that stops reading, then remove the task or cancel its request; assert bounded memory/queue size, closed runtime readers/channels and goroutine completion within the test deadline. Also assert that midstream errors do not attempt to replace already-sent plain-text headers with a JSON error status. This is an unimplemented design answer, not a claim that the current adapter has that complete policy.

These answers distinguish source reasoning from the scoped execution records linked in chapter 16. Proposed changes require separate implementation and validation.

Chapter 17 — worked answers

1. One stop

Four available tasks minus one leaves three; desired three minus remaining three gives zero unavailable. A second stop would leave two and violate MaxUnavailable zero. The replacement guard also limits old stops to the one healthy replacement. Both conditions in stopOldTasks matter; surge is a creation ceiling rather than independent authorization to remove.

2. Cohort-specific health

rolloutStateFor calls isAvailable for old tasks, accepting running/healthy, but increments newHealthy only for TaskHealthy. A running target occupies a slot without authorizing replacement. This prevents transferring the old cohort's availability predicate to the new cohort. It still relies on agent reports and does not independently prove routing.

3. Unhealthy is not failed

Only TaskFailed increments newFailed in rolloutStateFor. The agent reports unhealthy at its threshold; MemoryService converts that report to failed only when restartAllowed permits replacement. Otherwise the stored unhealthy target can stall progress until health recovers or an explicit failed report arrives. Observe actual task transitions and the deployment separately; do not infer a failed deployment from one HTTP probe result.

4. Identity survives only with its owner

The same live MemoryService retains activeRollback and returns that deployment on repeat. Capture the operation ID and unchanged version history. A fresh default server has new maps, so the previous ID/history

are absent; retry cannot establish that the original reverse transition was resumed. A durable-domain recovery protocol would need different source and process-level evidence.

Return to Chapter 17.

Chapter 18 — worked answers

1. Existing identity versus missing identity

DeleteService finds namespace-visible service identity before checking whether it is already deleting/deleted. Existing ownership can repeat safely, but a missing object cannot prove a previous workflow exists. The result is intentional idempotency with preserved identity, not a blanket success for every unknown request.

2. Agent owns runtime retry

The agent's desired-stop branch resolves the same owned object and repeats stop/remove. removeLocalContainer returns before removed reporting on failure, so finalization sees a task not yet removed. A second attempt can tolerate already-stopped state and finish removal. Evidence must distinguish the actual stop from the missing removal acknowledgement.

3. Metadata is not deletion authority alone

ContainerID can be stale or wrong. assignedContainerForRemoval checks managed identity and matching node/task labels, searches scoped labels when needed, and records drift instead of removing an unrelated object. Assert sentinel unchanged and record the mismatch; do not change its labels merely to satisfy cleanup.

4. Stopped is not removed; data is not the container

Default finalization requires ActualStatus removed, not stopped. Named volume data also has a lifecycle independent of task/container cleanup. Inspect each run-owned container, accept the removed task report,

verify final service status, and inspect only recorded lab volumes. Broad pruning risks unrelated data and does not establish the missing workflow acknowledgement.

Return to Chapter 18.

Chapter 19 — worked answers

1. Strict comparison and sampling

The implementation uses LastHeartbeatAt.Before(cutoff), not less-than-or-equal. Equality is therefore skipped. The next ticker can detect staleness, adding up to a cadence phase under normal scheduling; do not claim exact detection latency without observed timestamps and scheduling assumptions.

2. Decision completion differs from removal

nodeDrainStatusLocked increments RemainingTasks only for tasks still desired neither stopped nor removed. It can return complete after stop intent is set while an agent has not removed the container. Verify actual task removal and Docker identity before equating drain status with an empty host.

3. Conservative stateful policy is not fencing

Both get a node_lost condition. Stateless tasks become desired removed, actual failed and eligible for replacement; Stateful tasks retain their state in this branch. Neither proves exclusive runtime data access. A partitioned old process can keep running, and preserving a stateful task does not recover its data.

4. Evidence must span both owners

Use two disposable agents and an owned server or local proxy. Retain nodeA/taskA/containerA and nodeB/taskB/containerB plus UTC samples of heartbeat, central assignment and both direct endpoints. Pause only the owned agent, preserve old-container responses after replacement starts, then resume and inspect cleanup. That demonstrates the bounded scenario without assuming all partitions behave identically.

Return to Chapter 19.

Chapter 20 — worked answers

1. One node address is not two sockets

Resolve deduplicates IPv4 strings, so two tasks on one node contribute one answer. Their public ports are not represented in A records. The endpoint API retains task IDs and each published port; a client needs that data or an explicitly known common fixed port/gateway.

2. Separate predicates

includeTask returns false when DesiredStatus is not running, so HTTP discovery excludes the task. DNS Resolve checks only actual running/healthy in its task loop, so an otherwise resolvable node IPv4 may remain. This is a source-derived divergence requiring a matching fixture before being called an observed runtime incident.

3. IPv4 conversion is explicit

Resolve uses net.ParseIP(host).To4. A hostname is not resolved and IPv6-only data has no To4 representation, so it contributes no A answer. HTTP endpoint records can still contain the advertise string because their builder does not perform the same conversion. Different surfaces need not produce equivalent sets.

4. Keep three owners visible

Record endpoint API tasks/versions/ports, DNS answer/TTL/query time and exact application connection destination. Compare a fresh resolver query with the client's cached record and inspect whether the client reused an existing connection. Test the correct recorded host port independently. Do not infer an exact staleness bound from TTL alone, and do not change a production resolver to perform the experiment.

Return to Chapter 20.

Chapter 21 — worked answers

1. Two different listeners

AdvertiseAddress identifies the node/agent communication surface, not a workload listener. addressHost extracts its hostname; endpointURL combines that host with PublicHostPort. Using the agent port would send application requests to the wrong listener even when the node is healthy.

2. TLS scope is explicit

route.TLS adds router.TLS, while endpointURL always constructs an HTTP backend URL. Certificate provisioning and gateway-to-worker encryption are separate deployment responsibilities. The source supports client-facing TLS intent without proving either certificate issuance or internal encryption.

3. Observe each hop

Save the provider JSON and exact backend. Request the application's local published port from its worker, then from the gateway's network context, then request the gateway with the configured Host/path. Compare agent probe origin and gateway origin. A firewall or topology failure can preserve local health while blocking the gateway, so neither observation invalidates the other.

4. Extension acceptance needs ownership

For IPAM, crash after allocation but before task acknowledgement, restart with durable task identity, and verify address ownership is recovered without duplicate allocation or orphan leakage. A CNI variant similarly repeats ADD/DEL after response loss with pinned plugins and bounded execution. These are proposed extension tests, not mechanisms currently supplied by orch.

Return to Chapter 21.

Chapter 22 — worked answers

1. Five before the maximum, four after

Current three exceeds minimum two. ceil(3×90/60)=ceil 4.5=5; maximum four clamps to four. The result changes desired replicas only if a caller executes the controller and ScaleService accepts the change; no cluster-fit or traffic result follows automatically.

2. Memory owns cooldown

lastScaleAt is created empty in NewController and written after successful scaling. A replacement controller lacks the previous timestamp even if the same MemoryService persists. A fresh default server additionally loses service state. Durable cooldown requires another owner and recovery test, not a larger duration setting.

3. Declared fields and nil results are bounded evidence

StabilizationWindow is validated but unused in this controller, so it proves neither retained history nor downscale damping. Provider errors are handled as skip_metrics with nil return, so successful RunOnce may represent no decision. Inspect emitted decision/reason and provider timestamps.

4. Provider and composition are separate gates

Specify percentage units, task/version aggregation, freshness limits, missing samples, authenticated bounded queries and reproducible fixtures. Then verify the executable constructs that provider/controller and routes accepted decisions through admission/quota/maintenance. Default startup has no such construction, so valid YAML is intent without an active consumer.

Return to Chapter 22.

Chapter 23 — worked answers

1. Two middleware paths

User routes skip configured JWT verification when the secret is empty. Agent routes are excluded from user roles and then call authorizeAgentCredential in their own handlers. Absence of a user principal therefore does not waive node credential validation.

2. Static configuration overrides claims

When staticUsers is nonempty, the subject must be present; its configured role replaces Principal.Role and NamespaceRoles is cleared. A configured operator global role then determines authorization. Inspect effective configuration without exposing bearer material.

3. Rotation is a state mutation before acknowledgement

Heartbeat issues a new token and SetAgentCredential replaces AgentTokenHash/expiry before the response reaches the agent. Losing that response can leave the agent holding a hash-mismatching old token. Expiry/revocation/re-enrollment need separate recovery analysis; short lifetime alone does not solve the acknowledgement window.

4. Different trust properties

JWT/node tokens establish possession-based API authority, not encrypted transport or certificate-bound peer identity. Docker socket access permits powerful host runtime operations regardless of a viewer's limited API role. Mutual TLS and host containment need explicit additional mechanisms and evidence, neither supplied by token naming or a future interface.

Return to Chapter 23.

Chapter 24 — worked answers

1. Metadata and execution are different interfaces

Secret GET/list return SecretMetadata, not plaintext. resolveEnvLocked decrypts a namespace-scoped reference and puts it in the assigned task Env, which the authorized agent needs to create a container. Protecting the metadata interface does not imply the agent/Docker administrator cannot inspect execution values.

2. Integrity and custody have separate owners

Each Encrypt call gets a random GCM nonce, prepends it and authenticates KeyID as associated data. Decrypt checks expected key identity and GCM authentication. The wrong-key fixture proves rejection in that implementation; it does not prove secure key storage, external KMS, rotation or default process durability.

3. Explicit override wins

Loader order yields key C, assuming all values meet helper/validation rules. Redacted prints the sentinel for a nonempty secret, intentionally concealing which concrete value won. Use an in-memory fixture assertion for precedence and keep public evidence to configuration-source and pass/fail outcomes.

4. Observe three objects separately

Record encrypted record metadata/key identity, a boolean assigned-task canary check and container identity/start time. Updating stored secret can change future resolution without editing an existing container environment. A new envelope key can reject old ciphertext; a rotation extension needs re-encryption and old-key retirement evidence. Never dump real plaintext to prove either boundary.

Return to Chapter 24.

Chapter 25 - Worked Answers

1. Registry port versus tag

The parser compares the last colon with the last slash. In registry.example.test:5000/team/atlas:v2, the port colon occurs before a slash, while the v2 colon occurs after the final slash. Registry becomes registry.example.test:5000, name team/atlas and tag v2. No network lookup is needed. The parser's accepted digest/reference syntax is its own supported subset, not proof that the registry serves the artifact.

2. Synthetic digest scope

Syntactic acceptance does not establish that an image exists, that the caller can pull it, or that a trusted publisher signed it. It also does not establish vulnerability safety or a particular platform manifest. The test uses synthetic digest data only to exercise metadata/pinning decisions. An executable lab needs an available independently verified artifact and actual pull/inspection evidence; copying a repeated-character digest into a manifest would not meet that gate.

3. Parser-only resolution

ParserResolver.Resolve calls Parse without contacting a registry. A tag-only request therefore produces no digest, even when a real registry could resolve that tag. RequireDigest checks the resulting metadata and rejects it. A resolver interface and an injected fixed resolver test demonstrate extensibility and field flow, not a shipped remote adapter. Supply a real digest or separately implement/test a trusted remote resolver with authentication, platform selection and bounded errors.

4. Private pull evidence

Create a scoped robot credential without logging its password, reference its namespace-local ID in an ordinary service, and retain sanitized service/task IDs, requested/pinned image and actual Engine outcome.

Verify that the same available digest reaches pull and create. Inspect only safe metadata and clean up exact run-owned resources. JobSpec lacks imagePullSecret and ListAssignedTasks takes an earlier Job branch without ordinary service auth/environment resolution, so this observation cannot certify private Job pulls by analogy.

These answers use chapter 25 and its scoped claim/execution records. Design changes remain proposals until separately implemented and tested.

Chapter 26 - Worked Answers

1. Read-only root field path

ListAssignedTasks supplies the service SecurityContext; ensureTask converts it with dockerSecurityContext and sets ContainerSpec.Security; dockerContainerConfig sets HostConfig.ReadonlyRootfs. Bind mounts have separate read-only/rw modes, so a read-only root does not prohibit every writable mounted path. The mapping is inspected source evidence, not a real Engine inspection. Empty configured CapDrop defaults to NET_RAW in this baseline, not ALL.

2. Order before defaults

CreateService resolves image metadata and validates the candidate, then calls Admit before NormalizeServiceSpec applies resource defaults. RequireResourceRequests can therefore reject missing explicit CPU/memory even when defaults would later exist. Trace the actual method rather than treating the design document's normalized-candidate phrase as execution. A change to admit normalized values would alter policy semantics and needs its own tests.

3. Different rejection types

Malformed YAML/domain structure can fail before policy evaluation. A structurally valid privileged request under denying policy reaches admission and returns field/rule/message violations. Quota exhaustion occurs at the later projected-usage check and returns quota.Error, exposed as its own structured conflict. Only the policy denial should be interpreted as admission violations. Human Error() text is not the stable machine interface; use typed/structured fields.

4. Scope of passing tests

The focused tests cover selected ordinary service scale/rollout and evaluator rules. CreateJob has a separate path, and GitOpsStatus auto_revert restores stored desired state without re-entering Admit in that method. Neither path is certified by the service tests. Shared Docker workers still share kernel/network/daemon trust; accepted user/capability/filesystem options do not create a complete hostile-tenant sandbox. A broader claim needs caller inventory and explicit physical-boundary tests.

These answers use chapter 26 and its scoped claim/execution records. Design changes remain proposals until separately implemented and tested.

Chapter 27 - Worked Answers

1. Absent versus explicit context

Selected records whether the private context key exists. Matches returns true for any object namespace when no explicit selection exists, allowing internal inventories. With an explicit default selection, it compares normalized namespace and limits visibility to default. HTTP middleware supplies explicit selection, so a context.Background internal call is not equivalent to an authenticated HTTP request. Tests must preserve that distinction.

2. Forbidden versus absent

A principal with only a team-a role selecting team-b can fail authorization before object lookup and receive forbidden. A request authorized within explicit team-a that uses a team-b object ID reaches the namespace match and returns not-found. Authentication and malformed names can fail still earlier. Vary one layer at a time in a fixture; otherwise a denial does not prove the intended scoped lookup was exercised.

3. Secret reference scope

Service creation checks namespacedKey(team-b, secret-name). A team-a record cannot satisfy it, so the missing local reference returns invalid state rather than decrypting or borrowing the other record. Ordinary assignment later resolves using the same workload namespace. The executed fixture tests this with disposable values. It establishes logical reference isolation, not independent agent/host memory or physical storage isolation.

4. Destructive rollback and physical limits

The down migration deletes non-default services, tasks, deployments, events, audit rows, secrets and registry credentials plus related service versions before dropping namespace columns. It cannot preserve both

teams' data under old global uniqueness. Require a backup/restore plan and explicit data-loss review; the chapter does not execute that downgrade. Even a correct logical namespace model leaves shared workers, Docker authority, networking and physical volume naming as separate trust boundaries.

These answers use chapter 27 and its scoped claim/execution records. Design changes remain proposals until separately implemented and tested.

Chapter 28 - Worked Answers

1. Atlas arithmetic

Two replicas at250m and128Mi contribute500 millicores and268435456 bytes. The service count is one and replica count two. One positive published-port declaration contributes one, not two: addServiceUsage counts declarations separately from per-replica requested CPU/memory. A scale-zero service can still count as a service with positive port declarations. These are declared quantities, not measured process utilization or actual host socket counts.

2. Concurrent acceptance

Starting at two total replicas under maximum three, either single scale-to-two is valid and projects three. Both succeeding would project four. MemoryService holds its mutex across replacement usage, Check and mutation, and the executed fixture observes one success/one denial. The PostgreSQL path separately takes a namespace transaction advisory lock before usage/check and must retain it through mutation. That SQL mechanism is inspected here, not the default executable backend or an executed database quota race.

3. Cumulative runtime is sampled twice

The current report sums both snapshots and returns180 runtime seconds. Each snapshot contains lifetime-to-date for retained tasks, so the second includes the first minute again. The total is not120 unique elapsed/billable seconds. Requested CPU/memory totals are likewise sums of samples rather than integrated core-hours/byte-seconds. A downstream duration model must define interval integration or deltas, retention and missing samples instead of renaming the existing total.

4. Operational interpretation

Export a bounded namespace/time range, retain RFC3339 timestamps and state each column's unit. Use requested quantities for declared-footprint analysis, claim count for logical inventory and cumulative task duration only with a documented non-double-counting transformation. Do not infer resident memory, CPU time, stored bytes, pricing or invoices. Default memory snapshot lifetime and retention also limit history. Commercial accounting requires additional collection, durable records and independently validated definitions.

These answers use chapter 28 and its scoped claim/execution records. Design changes remain proposals until separately implemented and tested.

Chapter 29 - Worked Answers

1. Identity is not authorization

A digest can consistently identify malicious bytes. A trust decision needs accepted signature identity/key, subject binding and policy context, and even a valid trusted signature does not prove vulnerability absence or honest behavior. Baseline digest policy checks supplied metadata; it does not activate the signing/scanning designs. Keep registry access credentials separate from publisher trust and never send a private signing key merely to verify.

2. Three proposed points

Admission gives early feedback, rollout binds the selected version and agent verification protects the final delayed execution/bypass boundary. All must bind the same digest/namespace/policy and handle expiry. Those signature-verification hooks are proposed in the document, not active baseline calls. The actual baseline has image parsing/policy and ordinary service admission. The book fake-verdict contract tests only its own acceptance truth table.

3. Freshness changes without changed bytes

A new vulnerability database can identify a problem in an unchanged image. Record scanner/database version and expiry as well as digest, trusted producer and completion state. A completed old result is not automatically fresh evidence. The design proposes these records; the book does not run an external scanner or ingest an SBOM. Missing, pending, unavailable, expired and policy-violating results need distinct reasons and explicit fail behavior.

4. Exception ordering

If a pending-scan allow branch runs before subject/signature checks, a pending verdict for another namespace/digest/policy or an untrusted signature can be accepted. Keep binding/trust/expiry checks first. The executed test explicitly sets pending plus untrusted and expects denial, and the binding test varies namespace/digest/policy. Add combinations if widening the exception. allow_with_audit remains an obligation for a future caller, not an actual durable audit operation in this teaching function.

These answers use chapter 29 and its scoped claim/execution records. Design changes remain proposals until separately implemented and tested.

Chapter 30 – Worked Answers

1. Two revision domains

The Orch source SHA identifies the controller implementation, parser and mutation behavior. The local manifest repository's checkout commit identifies the intent applied by that implementation. Both are needed for reproducibility; a mutable branch name identifies neither exact bytes nor historical behavior. The teaching run records real full manifest commit IDs without pushing a repository or pretending they are app-source revisions.

2. Partial apply provenance

The first sorted file can already mutate its service before parsing the second fails. LastError records the sync failure, while LastRevision can still describe the earlier successful sync because revision update happens after the loop/prune. Inspect live service specs/managed markers and retained command output to reconstruct partial effects. A failed sync is not proof of no changes, and deterministic order is not a transaction. An atomic design needs an explicit validation/apply protocol.

3. Mutating status

GitOpsStatus can restore DesiredSpec, increment version and reconcile under auto_revert. Therefore calling it can change accepted intent. That branch does not freshly call admission/quota as ordinary ApplyService does, so changed constraints require separate reasoning/testing. The executed Atlas fixture shows warn retaining manual replicas two and auto_revert restoring desired zero. It does not prove a continuous passive drift observer or physical container equivalence.

4. Prune ownership test

Mark a previously managed same-name service with a different current source owner, remove its manifest from the first source, then sync with prune. The baseline tests prior/wanted name sets within namespace but does not verify current source ID/object ID before deletion. A safer design should reject ownership mismatch or require explicit transfer semantics. The passing fixture only proves a never-managed unrelated name survives; it does not prove same-name replacement/transfer safety.

These answers use chapter 30 and its scoped claim/execution records. Design changes remain proposals until separately implemented and tested.

Chapter 31 - Worked Answers

1. Attempt arithmetic

Attempts begins at one. With limit zero, first failure satisfies Attempts greater than limit and ends failed. With limit one, first failure creates attempt two, and its failure ends failed. Thus maximum total attempts are one and two. The helper immediately creates a new pending task and has no timer/backoff delay. Different task IDs preserve attempt history; this does not make repeated external effects idempotent.

2. Missing exit-code defaults

updateJobForTaskLocked defaults stopped to zero and failed to one when the pointer is absent. The former can produce succeeded; the latter enters retry/failure. These defaults interpret accepted reports, not physical proof that a process exited cleanly. A live lab must retain actual Engine exit evidence. The focused fixture injects explicit codes and establishes the selected state transitions only.

3. Backlog and Forbid

Between returns earliest due occurrences: minute one and minute two for a three-minute delay with limit two. If Forbid sees an active referenced pending/running Job at the start, it suppresses creation without advancing LastScheduleAt there. If none is active, the due loop can create several backlog Jobs in that same pass; it does not recalculate overlap after each. This is narrower than a universal no-overlap guarantee.

4. Checkpoint and DST evidence

Inject failure after CreateJob succeeds but before UpdateCronJob saves LastScheduleAt, then inspect next-pass behavior with active and completed same-name Jobs. Stronger scheduling needs atomic/deduplicated occurrence acceptance and external-effect identity, plus

durable restart state. For DST, specify an IANA zone and exact spring/fall transition, compute local/UTC occurrences with injected time and retain actual results. Ordinary UTC/Belgrade October13 arithmetic does not establish skipped/repeated-hour policy or exactly-once execution.

These answers use chapter 31 and its scoped claim/execution records. Design changes remain proposals until separately implemented and tested.

Chapter 32 - Worked Answers

1. Three storage identities

Claim name selects a namespace-local logical claim; VolumeID identifies metadata; VolumeName is what the agent forwards to Docker's name:target:mode bind. The current mapping does not add namespace or UUID to that physical name. Equal names in different logical namespaces on one daemon can therefore collide, despite scoped metadata. Use a unique run-owned name and actual mount inspection for live tests; do not infer physical storage isolation from secret-reference tests.

2. Conservative active attachment rule

For a new writable RWO mount without concurrent-writer permission, the helper rejects if any active attachment has the same VolumeID. It does not filter existing attachments by ReadOnly. A new reader skips the writable branch, but an existing reader can block a later writer. This qualifies the documentation's simpler readers-do-not-consume-lock statement. allow_concurrent_writers bypasses that branch without implementing application-level safe concurrent writes.

3. Logical lock versus lost node

The test proves one accepted metadata writer and later detached records under injected reports. A lost heartbeat does not prove the old host process stopped, and another host does not automatically possess the local volume's bytes. Metadata RWO is not distributed fencing, replication or backup. Safe failover requires actual revocation/process evidence and an explicit data recovery/migration plan outside the present MVP.

4. Retention and stale binding

Create an exact owned named volume with harmless sentinel data, replace only its run-owned container on the same node and inspect the same mount/data. State whether final cleanup retains or deletes that exact volume; never broad-prune. Separately capture a resolved mount before its first binding, bind the volume, then attempt later placement using the old snapshot. Assert latest topology is enforced or document failure. The helper checks mount.NodeID, so copied stale snapshots need explicit coverage before claiming every retry remains pinned.

These answers use chapter 32 and its scoped claim/execution records. Design changes remain proposals until separately implemented and tested.

Chapter 33 worked answers

These answers use baseline cbfc94ade5b57e449d6b73bf1485b1c5b1647022 and the chapter's linked execution records. Procedures remain scoped to disposable resources.

1. Existing composition determines capability

The local secret envelope is runtime-integrated: cmd/orch-server/main.go142–147 constructs it and supplies WithSecretEnvelope to MemoryService. The autoscaler MetricsProvider is an implemented interface, but main does not construct its controller. A seam's declaration alone cannot establish an active loop. Notification dispatch is another integrated boundary because main installs the dispatcher.

2. Reject an incomplete handshake

Copy adapter_contract_test.go to .book-labs/ch33 in the disposable module and run `go test ./.book-labs/ch33 -count=1 -v`. The recorded command passed replay, cancellation, cleanup, wrong-key rejection and four handshake cases. Removing task-idempotency from the compatible fixture must make validation fail before any runtime call; the separate handshake validator performs no effects. The exercise is a proposed contract, not an upstream RPC protocol or supervisor. The runtime fixture uses FakeRuntime and therefore creates no engine containers.

3. Unknown completion needs inspection

Use a stable logical task-create key across retries, with a specification fingerprint or conflict rule. The first request might have created the remote container before its response was lost. After process restart, query the underlying engine's managed association for that task before creating again. A new transport request ID is acceptable; a new logical operation key would defeat deduplication. The current fake replay test covers one fake instance and cannot prove this durable cross-process behavior.

4. Isolation does not remove storage authority

The unsafe assumption is that an external process ceases to be trusted merely because it is separate. Storage access can expose or destroy workload data. Acceptance must verify filesystem/volume scope, authenticated instance identity and socket ownership, and bounded calls/cancellation with recoverable uncertain completion. Add artifact provenance and redacted errors to the deployment review. A sandbox allowing unrestricted host paths cannot claim storage isolation; a crash-containment boundary is narrower than an authorization boundary.

See the corresponding chapter claim ledger for exact immutable source anchors and limitations.

Return to Chapter 33.

Chapter 34 worked answers

These answers use baseline cbfc94ade5b57e449d6b73bf1485b1c5b1647022 and the chapter's linked execution records. Procedures remain scoped to disposable resources.

1. A local map is not distributed authority

Each LocalElector owns its own mutex/map. Two different instances can each see an empty map and acquire scheduler. The one-leader test shares one elector, so its exclusion claim is scoped to that shared instance. PostgreSQL coordination needs the same authoritative database session space and matching name; separate primaries can each permit an owner.

2. Loss ordering is observable

Run `go test ./internal/leadership -count=1 -v` in the disposable copy. The recorded run passed. The wrapper acquires, the callback closes started, the test waits for started, then Lose closes the lease channel. The wrapper cancels the child context, the cooperative callback returns context.Canceled, and the metric becomes false. Channels establish ordering without timing sleeps. A callback ignoring its context is outside this success argument.

3. Done is not SQL unlock

postgres.go90–92 calls close on cancellation; close105–109 closes Done and returns the connection to the pool. A live pooled PostgreSQL session can keep its advisory lock. Release67–71 sees Done closed and returns without pg_advisory_unlock. The actual full-schema record observed successor blockage after cancellation/release and acquisition only after the original pool closed. This is a package-level session cleanup finding, not proof of a production HA deployment. Repair must explicitly unlock or terminate the relevant session without returning a still-locked live connection to general reuse.

4. Reject the memory-replica shortcut

Independent MemoryService processes have divergent records and credentials. First implement durable transactional API/controller parity, persistent cluster identity, one verified database primary, and readiness tied to that authority. Then wire per-controller leases with bounded reacquisition, cooperative cancellation, explicit connection cleanup and idempotent effects. Test in-flight effects and database partitions separately. An API health response cannot prove shared state or that an old leader's external request stopped.

See the corresponding chapter claim ledger for exact immutable source anchors and limitations.

Return to Chapter 34.

Chapter 35 worked answers

These answers use baseline cbfc94ade5b57e449d6b73bf1485b1c5b1647022 and the chapter's linked execution records. Procedures remain scoped to disposable resources.

1. Read the pinned ranges

Server0.3.0 advertises APIv1, agent minimum0.2.0/maximum-tested0.3.0, expected schema16, accepted schema15–16. Expected names the preferred current schema; accepted is the checker's range, not a proof that every package query succeeds at every value. The original fixture omitting migrations15/16 failed on agent_version introduced by16. Keep that observation separate from the declared range.

2. Parse the actual decision

Run the ch35 teaching fixture. Actual real Cobra execution returned no command error for0.1.0 while its JSON status was blocked. Automation must validate the JSON shape, reject status blocked, and explicitly decide whether warnings such as untested-newer require review. A zero exit code alone only establishes successful command/output handling here. The fixture's nodes are controlled inputs, not a live cluster inventory.

3. Partial upgrade is possible

Up applies each migration in its own transaction. If eleven fails after ten committed, status should retain the earlier versions and show the remaining work. Inspect the failing SQL, current/pending records, candidate binary compatibility and verified backup before retrying. Do not edit committed migration history or run down against live data. The actual owned database run applied all16, skipped them on repeated up, and rejected down without allow-down; it did not demonstrate destructive rollback safety.

4. Preserve unresolved state and trace policy

Old running tasks stay. Old failed tasks stay if unresolved; a qualifying running/healthy replacement at the same/newer version or cleanup desired status resolves them for age-based pruning. The recorded deterministic retention test checked these cases. Maintenance allows operations when no relevant enabled window exists; an unrelated window does not freeze the cluster. GitOpsRecords is configurable but the inspected prune method has no GitOps deletion loop. A field is not enforcement until its caller/mutation path exists.

See the corresponding chapter claim ledger for exact immutable source anchors and limitations.

Return to Chapter 35.

Chapter 36 worked answers

These answers use baseline cbfc94ade5b57e449d6b73bf1485b1c5b1647022 and the chapter's linked execution records. Procedures remain scoped to disposable resources.

1. Read the fake's contract

Task-key replay and operation history support agent idempotency/order assertions. FakeRuntime does not enforce kernel cgroups or real port binding, so a successful fake start cannot prove memory or CPU limits. Assert supplied ContainerSpec fields and separately inspect a real engine when claiming enforcement.

2. Reproduce a property, not a score

Run `go test ./internal/scheduler -run '^TestSchedulerRandomizedInvariants/seed_42$' -count=1 -v`. The batch's actual recorded run included all five seeds and passed. Seed42 must accompany the exact generator revision because changing generated fields/order changes the resulting fixture. For occupied-port coverage, add an existing active task reserving a fixed TCP port on one node and a pending service requesting it; assert no assignment reuses that node/port pair. Do not call the scoring helper again as the oracle. The baseline randomized fixture has no occupied-task inventory.

3. Fuzz scope is callback scope

Run the exact FuzzPortSpec target for two seconds with one worker. The recorded campaign performed42,919 executions and passed. Its callback discards values/errors, so panic resistance is the main assertion; accepted-invalid semantics can remain undetected. Sixty-three newly interesting inputs report coverage exploration, not sixty-three bugs. No crash occurred, so no minimized failing case is invented. The documentation's FuzzParsePortSpec example does not match the actual declaration.

4. Observe every handoff

Submit explicit CPU/memory through the normal API path, inspect stored service values, inspect the delivered agent DTO, assert the Runner's created ContainerSpec, then inspect the actual engine limits. Keep identities tied to one owned task/container. The actual boundary experiment found direct adapter limits present and normal-path limits zero. Testing only a manually populated direct-adapter spec would pass while missing the dropped DTO fields. Preserve the failing upstream boundary; a regression fix requires separately authorized application work.

See the corresponding chapter claim ledger for exact immutable source anchors and limitations.

Return to Chapter 36.

Chapter 37 worked answers

These answers use baseline cbfc94ade5b57e449d6b73bf1485b1c5b1647022 and the chapter's linked execution records. Procedures remain scoped to disposable resources.

1. Name the modeled boundary

Rename agent-restart chaos to repeated running report against retained MemoryService. Rename server-restart chaos to rollout controller reconstruction over retained store. The first does not kill an agent; the second does not create a fresh server/control plane. Their useful assertions are report idempotency and recomputation, not durable process recovery.

2. Unknown existence prevents speculative create

Run the chapter's exact selected test command. The actual record passed all eight chaos subcases and selected agent/fake tests. The unavailable-runtime case expects inspect:container-1,list,list and zero status reports. There is no create in that operation history. Both inspect and list fail, so reality is unknown rather than confirmed absent. Creating a replacement could duplicate a still-running object.

3. Cleanup on return is not fencing during partition

A node partitions while its container continues. The control plane eventually marks it lost, removes its active assignment and replaces a stateless workload elsewhere. Until the old node reconnects and consumes cleanup, both physical workloads may exist. Returned-node reconciliation lists the managed unassigned old object, stops/removes it and reports removed. The fake test observes that cleanup sequence; it does not observe exclusive service traffic or storage fencing throughout the partition.

4. Match state owner to recovery claim

Reconstructed controllers can recompute from retained records. The combined actual experiment restarted the binary and found services/tasks/nodes absent while an exact manually associated nginx container survived. Durable rollout recovery requires a server actually composing persistent transactional state, fresh-process reads of deployment/task/version records, and observed recovery/cleanup of corresponding engine objects. Capture initial IDs, UTC transitions and final absence. Remove only owned exact objects/databases; never sweep unrelated containers. A healthy endpoint is insufficient recovery evidence.

See the corresponding chapter claim ledger for exact immutable source anchors and limitations.

Return to Chapter 37.

Chapter 38 worked answers

These answers use baseline cbfc94ade5b57e449d6b73bf1485b1c5b1647022 and the chapter's linked execution records. Procedures remain scoped to disposable resources.

1. Follow the producer and consumer

instrumentedRuntime.measure increments operation attempts and returned errors. Agent registry exposes docker_operation_errors_total with operation pull_image. prometheus.yml scrapes the agent job. DockerOperationErrorsHigh sums its five-minute rate above0.1for10minutes, and the Docker Operation Errors dashboard groups the rate by operation. The teaching fixture actually executed the wrapped fake failure and registry exposition; scrape/rule/panel were inspected, not live-evaluated.

2. Requests and distinct transitions differ

api.go701–702 increments after each accepted request whose requested status is failed. MemoryService can accept a duplicate failed report without emitting another transition event. Therefore the metric may count two accepted reports for one task failure. The source-inspected semantics support this distinction; do not invent a distinct-task cardinality guarantee from the metric's name.

3. Rename proxies honestly

Task State Counts should read cumulative reported task state changes. Node Status Counts should read agent scrape reachability. The former sums task_state_changes_total; the latter counts up values. Authoritative current counts need current task/node state from a verified control-plane inventory or explicitly maintained current-state gauges. Counter subtraction cannot generally reconstruct state after arbitrary transitions or restarts. Rollout requests minus failures likewise is not terminal-success inventory.

4. Test temporal and absent-data behavior

Use an owned Prometheus instance with exact job labels and rule version. Inject a short burst, observe samples/rate, and verify the ten-minute hold does not fire for an insufficient sustained condition. Then inject sustained qualifying error rate and observe pending/firing/recovery timestamps. Separately stop the agent scrape and inspect up plus heartbeat-series absence. A single fixture increment supplies no rate history or evaluator time. Retain workload recovery evidence independently from alert disappearance; no such live rule run was claimed in this chapter.

See the corresponding chapter claim ledger for exact immutable source anchors and limitations.

Return to Chapter 38.

Chapter 39 worked answers

These answers use baseline cbfc94ade5b57e449d6b73bf1485b1c5b1647022 and the chapter's linked execution records. Procedures remain scoped to disposable resources.

1. Identify the timed work

average_convergence_time averages initial service creation plus synthetic running reports in MemoryService. It excludes engine image pull/start, application health, HTTP/TLS and durable database work. SchedulerIterations counts simulation ticks; ReconciliationIterations counts initial service creations. Names alone do not prove package controller invocations. The observed100-service run's323records include historical failures/replacements rather than323desired live replicas.

2. Repeat bounded samples

Use the exact bounded-benchmarks record: Go1.25.0, Linuxamd64, CPUcount2,100ms, count3, benchmark memory reporting. Scheduler Plan's observed samples were11,991,457/12,012,169/12,050,878ns per operation, each718,664bytes and630allocations. The fixture has100ready nodes and1000pending tasks with fixed requests. Median/range describe these samples only. Shared-host variation and three short samples do not justify a confidence interval or production cluster limit. Correctness properties must remain intact when comparing optimizations.

3. Separate listener does not mean authenticated

NewDebugHandler registers pprof without its own auth middleware. Default loopback reduces exposure, but configurable ORCH_DEBUG_ADDR can broaden it. Restrict management access, bound captures, and treat profiles as sensitive. The chapter captured a synthetic benchmark profile without opening a listener. Prebuilt go tool pprof was absent; source-built go run cmd/pprof succeeded separately, with a short120ms CPU sample total.

4. Build a different production experiment

Pin images/cache state, services/replicas, update rate, constraints, authoritative durable store/schema, network path and readiness criterion. Measure API acceptance, scheduler computation, actual runtime start and application health separately. Include retained task/event history, database errors, partitions/recovery and exact cleanup. The tagged thousand-service test passed only task-count/iteration assertions with fake transitions. Running PostgreSQL beside that command does not make it a persistence benchmark. Record unresolved work rather than extrapolating a safe production maximum.

See the corresponding chapter claim ledger for exact immutable source anchors and limitations.

Return to Chapter 39.

Chapter 40 worked answers

These answers use baseline cbfc94ade5b57e449d6b73bf1485b1c5b1647022 and the chapter's linked execution records. Procedures remain scoped to disposable resources.

1. Preserve separate coverage lanes

The ungated baseline and race runs passed with explicit live integration skips. The original PostgreSQL fixture failed two store tests because it stopped at migration14. The full-schema extension applied16and verified owned store/restore behavior. The guarded Docker harness executed real handler/Runner/engine deploy,logs,scale/delete with bounded ownership. These are complementary records; none erases another lane's skips or failures. A passing inventory check only joins evidence.

2. Configuration does not compose missing behavior

Restart=always restarts a process but does not persist its maps. DATABASE_URL does not replace the default MemoryService. Mounted certificates/TLS environment placeholders do not make the current public listener consume native TLS; documented termination needs an actual proxy and verified configuration. Docker group/socket authority remains powerful despite a non-root service identity and filesystem hardening. No systemd units or production Compose deployment were installed in this chapter.

3. Restore the owner that actually persisted data

Use new exact owned source/target databases, verify their absence before creation, apply pinned migrations, insert synthetic canary and selected records, dump, restore into the separate target, compare expected values and reject a wrong key. Preserve actual command/UTC/IDs and clean only those databases/containers. Back up the envelope key/configuration separately through secure owner procedures. The

actual full-schema drill passed these package-level checks. It cannot restore prior default MemoryService records, Docker volumes, running containers, JWT keys or TLS materials that were absent from the dump.

4. A dossier is not deployment approval

Run verify_dossier.py. The corrected record passed its source/status/output-presence inventory and explicitly returned production_decision not approved. A sensible next guarantee is durable fresh-server recovery: implement transactional server composition, inject a partial-write failure, restart a real process, compare service/task/version/credential/event identities and clean owned runtime objects. Reject if any invariant or recovery evidence fails. Do not promote readiness because the JSON inventory exists; acceptance must observe the new behavior and retain leadership/resource/trust/network gaps until independently resolved.

See the corresponding chapter claim ledger for exact immutable source anchors and limitations.

Return to Chapter 40.

Atlas proposal: durable control plane before replicated APIs

Status: design proposal, not upstream wiring or a completed HA deployment.

Decision

Do not place a load balancer in front of independent MemoryService processes. First implement one durable control-plane composition with equivalent transactional API/controller semantics. Bind all replicas to one verified database primary and a persistent Atlas cluster identity; reject mismatches before readiness.

Give scheduler, reconciler, rollout, node monitor and autoscaler separate cluster-qualified leadership names. The baseline hashes only `orch:` plus controller name; cluster namespacing is new design. Use dedicated sessions with explicit unlock or verified session termination on cancellation and error. Do not return a live session with an advisory lock to a general pool. A connectivity probe is not a fencing token.

Wrap all mutating loops, pass the lease context into every operation, bound request duration and in-flight work, and wait for the old callback to exit before reporting successful handover. Add acquisition retries with bounded jitter outside baseline RunWithLeadership, which acquires once. Persist idempotency keys/optimistic task guards and event writes transactionally. Decide how database failover fences old writers and prohibit dual primaries.

Acceptance evidence still required

- Concurrent replicated API mutations against durable state retain task/event invariants.
- Lease cancellation, callback exit, actual database unlock and successor acquisition have separate observations.

- Database partition fails durable readiness and stops new mutating work; reconnect recomputes state.
- In-flight external effects remain discoverable and idempotent after uncertain completion.
- Cluster identity mismatch and dual-primary topology are rejected operationally; no claim of independent distributed consensus.
- Fresh server processes read the same persisted services, credentials, tasks and events after restart.
- Backup/restore preserves required state and keys, with measured recovery and cleanup of owned resources.

Rejected shortcut

Two running HTTP endpoints plus a green health response do not establish shared active state, exclusive work or durable recovery.