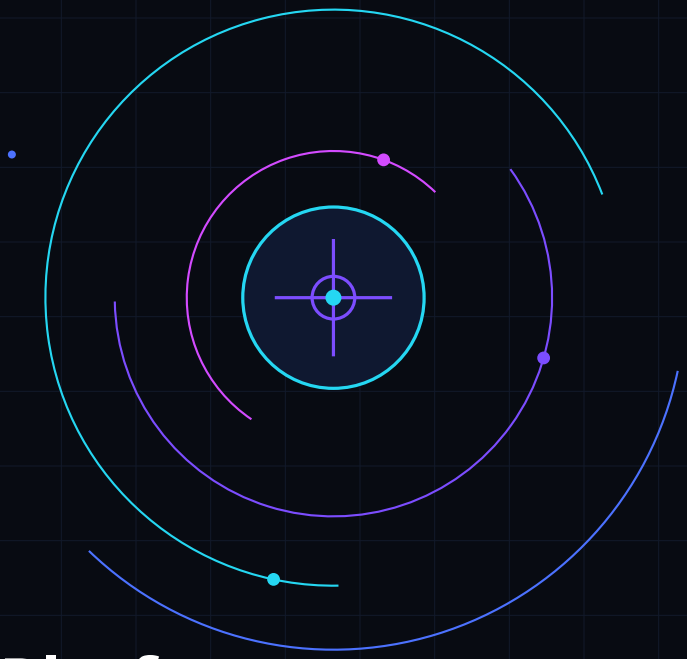

CANONICAL LIVING OVERVIEW GUIDE

# Internal Developer Platforms, GitOps, and Policy as Code

A platform-engineering guide covering platform-as-product, developer portals, catalogs, golden paths, control planes, self-service APIs, Crossplane, GitOps, Argo CD, Flux, policy as code, OPA, Kyverno, provenance, observability, adoption, and governance.

PERMANENT ID
MYTHOS-FG-CLD-0003

PUBLICATION CLASS
FIELD GUIDE

VERSION / STATUS
1.0.0-candidate / CANDIDATE

PERMANENT PUBLICATION IDENTITY

# Internal Developer Platforms, GitOps, and Policy as Code

Document ID
MYTHOS-FG-CLD-0003

Version
1.0.0-candidate

Status
Candidate Focused Field Guide

Review date
2026-10-26

Publication position
This is a living field guide. It is designed for frequent revision while retaining a stable permanent identity. Candidate status means the content is ready for structured use and review but is not represented as an external certification or authority publication.

| LEVEL | CLASS | CADENCE | EVIDENCE |
|---|---|---|---|
| L1-L4 | FIELD GUIDE | QUARTERLY | SOURCE-BOUND |
| Foundational through frontier | Practical and educational | Interim updates as needed | Limitations remain visible |

## Reader orientation

| Dimension | Engineering position |
|---|---|
| Primary domain | MYTHOS-CLD - Cloud, Platform & Distributed Systems |
| Secondary domain | MYTHOS-SWE; MYTHOS-SEC; MYTHOS-DAT |
| Audience | Platform engineers, developer-experience teams, SREs, cloud teams, security teams, and engineering leaders. |
| Prerequisites | Software delivery, Kubernetes, cloud, APIs, identity, infrastructure as code, CI/CD, and security. |
| Publisher / owner | Mythos Systems / Aaron Stovall |
| Public classification | Public technical guidance |

## Expected outcomes

Design internal platforms as products with defined users, outcomes, interfaces, ownership, and lifecycle.

Expose safe self-service through catalogs, templates, APIs, control planes, and golden paths.

Implement GitOps reconciliation with clear source authority, drift handling, promotions, and rollback.

Apply policy as code across source, CI, artifacts, infrastructure, admission, runtime, and exceptions.

Measure adoption, developer outcomes, reliability, security, cost, and platform effectiveness.

# How to use this guide

Read the guide as a progressive system rather than a disconnected encyclopedia. Foundational material establishes the mental model; practitioner sections convert it into repeatable implementation and operations; advanced sections expose architecture, scale, security, and failure behavior; frontier sections describe technologies whose evidence or maturity is still evolving.

Suggested reading path
New practitioners should follow the chapters in order and complete the laboratories. Experienced engineers may begin with the architecture, failure, validation, or frontier sections, then use the related-document map to deepen a specific topic.

## Learning and assurance model

| Level | Reader outcome |
|---|---|
| L1 - Foundational | Terminology, first principles, component roles, packet or state flow, and simple working examples. |
| L2 - Practitioner | Implementation, configuration, automation, testing, troubleshooting, and operational evidence. |
| L3 - Advanced | Architecture, scale, concurrency, resilience, threat models, performance, and complex migrations. |
| L4 - Frontier | Emerging systems, research directions, technical uncertainty, readiness gates, and adoption signals. |

# Contents

CHAPTER 01

# Platform product strategy and operating model

Define the platform's customers, promises, boundaries, and success measures.

Chapter operating position
Define the platform's customers, promises, boundaries, and success measures.

## 1.1 Users and journeys

Users and journeys must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Map developer, data, AI, operations, security, product, and support needs across delivery. The design should identify authoritative state, workload identities, typed contracts, resource and token budgets, trust boundaries, failure domains, telemetry, rollback, and acceptance evidence.

This guide traces the subject through the developer intent, catalog, template or API, policy, Git source, reconciliation, platform resource, evidence, and user outcome chain. A running agent, reconciled object, successful API call, healthy proxy, or completed function is not proof of the intended outcome. Desired state must be compared with applied state, external effects, negative tests, degraded behavior, causal evidence, cost, and user or mission results.

Production engineering begins with a minimal known-good control plane and explicit invariants. Add one worker class, model route, controller, policy, event source, gateway, or platform feature at a time; preserve before-state evidence; test authorized and prohibited behavior; inject realistic failures; and retain a tested compensation or rollback path.

Troubleshooting locates the first divergence from the expected state machine. Account for stale watches, leases, retries, eventual consistency, duplicated work, cache state, identity rotation, partial external effects, version skew, prompt injection, overload, network partitions, and missing telemetry. Closure requires causal explanation, reproducibility, validated recovery, and residual limitations.

### Engineering practices

- Define authority, ownership, desired state, dependencies, limits, and evidence for users and journeys.
- Use typed APIs, stable identities, explicit state machines, idempotency, and bounded effects.
- Separate control plane, execution plane, data plane, evidence plane, and human decision authority.
- Test nominal, prohibited, overloaded, partitioned, partially failed, recovery, upgrade, and rollback cases.
- Preserve configuration, policy, model, source, trace, event, artifact, approval, and user-outcome evidence.
- Scale through quotas, queues, backpressure, fairness, isolation, canaries, and deterministic fan-in.

### Failure patterns

- Treating users and journeys as a feature rather than an end-to-end stateful system.
- Scaling worker count or deployment scope before bounding queues, budgets, retries, and fan-in.
- Trusting controller, agent, or proxy status without validating external state and user outcome.
- Sharing credentials, workspaces, caches, or context across tenants and roles without explicit policy.
- Allowing AI-generated plans, policies, or changes to bypass deterministic validation and human authority.
- Declaring recovery after process restart without reconciling orphan effects, leases, artifacts, and evidence.

### Validation and evidence

| Method | Expected evidence |
| --- | --- |
| Examine | Requirements, APIs, identities, state machines, policies, versions, resources, and dependencies for users and journeys. |
| Observe | Queue, task, controller, model, tool, route, proxy, event, resource, trace, artifact, and user-visible state. |
| Test | Expected, prohibited, malformed, duplicate, overloaded, partitioned, failed, recovery, upgrade, and rollback conditions. |
| Measure | Task success, reliability, latency, throughput, fairness, cost, token and compute use, blast radius, and operator burden. |
| Reconcile | Desired and planned state against actual cloud, cluster, agent, network, data, external-effect, and evidence state. |

## 1.2 Platform capabilities and contracts

Platform capabilities and contracts must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Define environments, runtime, data, identity, delivery, observability, security, and support services. The design should identify authoritative state, workload identities, typed contracts, resource and token budgets, trust boundaries, failure domains, telemetry, rollback, and acceptance evidence.

This guide traces the subject through the developer intent, catalog, template or API, policy, Git source, reconciliation, platform resource, evidence, and user outcome chain. A running agent, reconciled object, successful API call, healthy proxy, or completed function is not proof of the intended outcome. Desired state must be compared with applied state, external effects, negative tests, degraded behavior, causal evidence, cost, and user or mission results.

Production engineering begins with a minimal known-good control plane and explicit invariants. Add one worker class, model route, controller, policy, event source, gateway, or platform feature at a time; preserve before-state evidence; test authorized and prohibited behavior; inject realistic failures; and retain a tested compensation or rollback path.

Troubleshooting locates the first divergence from the expected state machine. Account for stale watches, leases, retries, eventual consistency, duplicated work, cache state, identity rotation, partial external effects, version skew, prompt injection, overload, network partitions, and missing telemetry. Closure requires causal explanation, reproducibility, validated recovery, and residual limitations.

### Engineering practices

- Define authority, ownership, desired state, dependencies, limits, and evidence for platform capabilities and contracts.
- Use typed APIs, stable identities, explicit state machines, idempotency, and bounded effects.
- Separate control plane, execution plane, data plane, evidence plane, and human decision authority.
- Test nominal, prohibited, overloaded, partitioned, partially failed, recovery, upgrade, and rollback cases.
- Preserve configuration, policy, model, source, trace, event, artifact, approval, and user-outcome evidence.
- Scale through quotas, queues, backpressure, fairness, isolation, canaries, and deterministic fan-in.

### Implementation sketch

```
platform_request:
  api: ApplicationEnvironment/v2
  owner: team-payments
  environment: production
  composition:
    - kubernetes-namespace
    - workload-identity
    - database
    - gateway-route
    - telemetry
  policies:
    - signed-artifacts
    - restricted-pods
    - encrypted-data
  gitops:
    repository: environments-prod
    promotion: pull-request
  evidence:
    require: [policy-tests, drift-check, rollback-plan]
```

### Failure patterns

- Treating platform capabilities and contracts as a feature rather than an end-to-end stateful system.
- Scaling worker count or deployment scope before bounding queues, budgets, retries, and fan-in.
- Trusting controller, agent, or proxy status without validating external state and user outcome.
- Sharing credentials, workspaces, caches, or context across tenants and roles without explicit policy.
- Allowing AI-generated plans, policies, or changes to bypass deterministic validation and human authority.
- Declaring recovery after process restart without reconciling orphan effects, leases, artifacts, and evidence.

### Validation and evidence

| Method | Expected evidence |
| --- | --- |
| Examine | Requirements, APIs, identities, state machines, policies, versions, resources, and dependencies for platform capabilities and contracts. |
| Observe | Queue, task, controller, model, tool, route, proxy, event, resource, trace, artifact, and user-visible state. |
| Test | Expected, prohibited, malformed, duplicate, overloaded, partitioned, failed, recovery, upgrade, and rollback conditions. |
| Measure | Task success, reliability, latency, throughput, fairness, cost, token and compute use, blast radius, and operator burden. |
| Reconcile | Desired and planned state against actual cloud, cluster, agent, network, data, external-effect, and evidence state. |

## 1.3 Team topology and ownership

Team topology and ownership must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Assign platform product, engineering, SRE, security, enablement, service teams, and governance. The design should identify authoritative state, workload identities, typed contracts, resource and token budgets, trust boundaries, failure domains, telemetry, rollback, and acceptance evidence.

This guide traces the subject through the developer intent, catalog, template or API, policy, Git source, reconciliation, platform resource, evidence, and user outcome chain. A running agent, reconciled object, successful API call, healthy proxy, or completed function is not proof of the intended outcome. Desired state must be compared with applied state, external effects, negative tests, degraded behavior, causal evidence, cost, and user or mission results.

Production engineering begins with a minimal known-good control plane and explicit invariants. Add one worker class, model route, controller, policy, event source, gateway, or platform feature at a time; preserve before-state evidence; test authorized and prohibited behavior; inject realistic failures; and retain a tested compensation or rollback path.

Troubleshooting locates the first divergence from the expected state machine. Account for stale watches, leases, retries, eventual consistency, duplicated work, cache state, identity rotation, partial external effects, version skew, prompt injection, overload, network partitions, and missing telemetry. Closure requires causal explanation, reproducibility, validated recovery, and residual limitations.

### Engineering practices

- Define authority, ownership, desired state, dependencies, limits, and evidence for team topology and ownership.
- Use typed APIs, stable identities, explicit state machines, idempotency, and bounded effects.
- Separate control plane, execution plane, data plane, evidence plane, and human decision authority.
- Test nominal, prohibited, overloaded, partitioned, partially failed, recovery, upgrade, and rollback cases.
- Preserve configuration, policy, model, source, trace, event, artifact, approval, and user-outcome evidence.
- Scale through quotas, queues, backpressure, fairness, isolation, canaries, and deterministic fan-in.

### Failure patterns

- Treating team topology and ownership as a feature rather than an end-to-end stateful system.
- Scaling worker count or deployment scope before bounding queues, budgets, retries, and fan-in.
- Trusting controller, agent, or proxy status without validating external state and user outcome.

Sharing credentials, workspaces, caches, or context across tenants and roles without explicit policy.

Allowing AI-generated plans, policies, or changes to bypass deterministic validation and human authority.

Declaring recovery after process restart without reconciling orphan effects, leases, artifacts, and evidence.

### Validation and evidence

| Method | Expected evidence |
|---|---|
| Examine | Requirements, APIs, identities, state machines, policies, versions, resources, and dependencies for team topology and ownership. |
| Observe | Queue, task, controller, model, tool, route, proxy, event, resource, trace, artifact, and user-visible state. |
| Test | Expected, prohibited, malformed, duplicate, overloaded, partitioned, failed, recovery, upgrade, and rollback conditions. |
| Measure | Task success, reliability, latency, throughput, fairness, cost, token and compute use, blast radius, and operator burden. |
| Reconcile | Desired and planned state against actual cloud, cluster, agent, network, data, external-effect, and evidence state. |

## 1.4 Outcome and adoption metrics

Outcome and adoption metrics must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Measure lead time, cognitive load, reliability, security, support, self-service, and retention. The design should identify authoritative state, workload identities, typed contracts, resource and token budgets, trust boundaries, failure domains, telemetry, rollback, and acceptance evidence.

This guide traces the subject through the developer intent, catalog, template or API, policy, Git source, reconciliation, platform resource, evidence, and user outcome chain. A running agent, reconciled object, successful API call, healthy proxy, or completed function is not proof of the intended outcome. Desired state must be compared with applied state, external effects, negative tests, degraded behavior, causal evidence, cost, and user or mission results.

Production engineering begins with a minimal known-good control plane and explicit invariants. Add one worker class, model route, controller, policy, event source, gateway, or platform feature at a time; preserve before-state evidence; test authorized and prohibited behavior; inject realistic failures; and retain a tested compensation or rollback path.

Troubleshooting locates the first divergence from the expected state machine. Account for stale watches, leases, retries, eventual consistency, duplicated work, cache state, identity rotation, partial external effects, version skew, prompt injection, overload, network partitions, and missing telemetry. Closure requires causal explanation, reproducibility, validated recovery, and residual limitations.

### Engineering practices

Define authority, ownership, desired state, dependencies, limits, and evidence for outcome and adoption metrics.

Use typed APIs, stable identities, explicit state machines, idempotency, and bounded effects.

Separate control plane, execution plane, data plane, evidence plane, and human decision authority.

Test nominal, prohibited, overloaded, partitioned, partially failed, recovery, upgrade, and rollback cases.

Preserve configuration, policy, model, source, trace, event, artifact, approval, and user-outcome evidence.

Scale through quotas, queues, backpressure, fairness, isolation, canaries, and deterministic fan-in.

### Failure patterns

Treating outcome and adoption metrics as a feature rather than an end-to-end stateful system.

Scaling worker count or deployment scope before bounding queues, budgets, retries, and fan-in.

Trusting controller, agent, or proxy status without validating external state and user outcome.

Sharing credentials, workspaces, caches, or context across tenants and roles without explicit policy.

Allowing AI-generated plans, policies, or changes to bypass deterministic validation and human authority.

Declaring recovery after process restart without reconciling orphan effects, leases, artifacts, and evidence.

## Validation and evidence

| Method | Expected evidence |
| --- | --- |
| Examine | Requirements, APIs, identities, state machines, policies, versions, resources, and dependencies for outcome and adoption metrics. |
| Observe | Queue, task, controller, model, tool, route, proxy, event, resource, trace, artifact, and user-visible state. |
| Test | Expected, prohibited, malformed, duplicate, overloaded, partitioned, failed, recovery, upgrade, and rollback conditions. |
| Measure | Task success, reliability, latency, throughput, fairness, cost, token and compute use, blast radius, and operator burden. |
| Reconcile | Desired and planned state against actual cloud, cluster, agent, network, data, external-effect, and evidence state. |

# Chapter 01 laboratory and review

Practical laboratory
Interview three developer archetypes and create a platform product charter with measurable outcomes.

## Laboratory evidence package

- Architecture or topology showing boundaries, identities, dependencies, and authoritative inputs.
- Versioned configuration or code with explanatory comments and reproducible setup instructions.
- Nominal-path validation plus at least two injected failure or boundary conditions.
- Structured logs, traces, metrics, packet captures, test output, or other appropriate evidence.
- A concise decision record covering findings, limitations, residual risks, and next actions.

## Frontier extension

Explore adaptive platforms that personalize interfaces without fragmenting control.

Review gate
Advance only when another engineer can reproduce the result, explain the architecture, identify the failure boundary, and distinguish measured evidence from assumptions or projections.

CHAPTER 02

# Developer portal, catalog, and ownership graph

Make software, teams, dependencies, health, and actions discoverable.

Chapter operating position
Make software, teams, dependencies, health, and actions discoverable.

## 2.1 Catalog entities and metadata

Catalog entities and metadata must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Represent components, systems, APIs, resources, domains, users, groups, owners, lifecycle, and links. The design should identify authoritative state, workload identities, typed contracts, resource and token budgets, trust boundaries, failure domains, telemetry, rollback, and acceptance evidence.

This guide traces the subject through the developer intent, catalog, template or API, policy, Git source, reconciliation, platform resource, evidence, and user outcome chain. A running agent, reconciled object, successful API call, healthy proxy, or completed function is not proof of the intended outcome. Desired state must be compared with applied state, external effects, negative tests, degraded behavior, causal evidence, cost, and user or mission results.

Production engineering begins with a minimal known-good control plane and explicit invariants. Add one worker class, model route, controller, policy, event source, gateway, or platform feature at a time; preserve before-state evidence; test authorized and prohibited behavior; inject realistic failures; and retain a tested compensation or rollback path.

Troubleshooting locates the first divergence from the expected state machine. Account for stale watches, leases, retries, eventual consistency, duplicated work, cache state, identity rotation, partial external effects, version skew, prompt injection, overload, network partitions, and missing telemetry. Closure requires causal explanation, reproducibility, validated recovery, and residual limitations.

### Engineering practices

- Define authority, ownership, desired state, dependencies, limits, and evidence for catalog entities and metadata.
- Use typed APIs, stable identities, explicit state machines, idempotency, and bounded effects.
- Separate control plane, execution plane, data plane, evidence plane, and human decision authority.
- Test nominal, prohibited, overloaded, partitioned, partially failed, recovery, upgrade, and rollback cases.
- Preserve configuration, policy, model, source, trace, event, artifact, approval, and user-outcome evidence.
- Scale through quotas, queues, backpressure, fairness, isolation, canaries, and deterministic fan-in.

### Failure patterns

- Treating catalog entities and metadata as a feature rather than an end-to-end stateful system.
- Scaling worker count or deployment scope before bounding queues, budgets, retries, and fan-in.
- Trusting controller, agent, or proxy status without validating external state and user outcome.
- Sharing credentials, workspaces, caches, or context across tenants and roles without explicit policy.
- Allowing AI-generated plans, policies, or changes to bypass deterministic validation and human authority.
- Declaring recovery after process restart without reconciling orphan effects, leases, artifacts, and evidence.

### Validation and evidence

| Method | Expected evidence |
| --- | --- |
| Examine | Requirements, APIs, identities, state machines, policies, versions, resources, and dependencies for catalog entities and metadata. |
| Observe | Queue, task, controller, model, tool, route, proxy, event, resource, trace, artifact, and user-visible state. |
| Test | Expected, prohibited, malformed, duplicate, overloaded, partitioned, failed, recovery, upgrade, and rollback conditions. |
| Measure | Task success, reliability, latency, throughput, fairness, cost, token and compute use, blast radius, and operator burden. |
| Reconcile | Desired and planned state against actual cloud, cluster, agent, network, data, external-effect, and evidence state. |

## 2.2 Source of truth and ingestion

Source of truth and ingestion must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Use source-controlled metadata, providers, processors, validation, reconciliation, and conflict handling. The design should identify authoritative state, workload identities, typed contracts, resource and token budgets, trust boundaries, failure domains, telemetry, rollback, and acceptance evidence.

This guide traces the subject through the developer intent, catalog, template or API, policy, Git source, reconciliation, platform resource, evidence, and user outcome chain. A running agent, reconciled object, successful API call, healthy proxy, or completed function is not proof of the intended outcome. Desired state must be compared with applied state, external effects, negative tests, degraded behavior, causal evidence, cost, and user or mission results.

Production engineering begins with a minimal known-good control plane and explicit invariants. Add one worker class, model route, controller, policy, event source, gateway, or platform feature at a time; preserve before-state evidence; test authorized and prohibited behavior; inject realistic failures; and retain a tested compensation or rollback path.

Troubleshooting locates the first divergence from the expected state machine. Account for stale watches, leases, retries, eventual consistency, duplicated work, cache state, identity rotation, partial external effects, version skew, prompt injection, overload, network partitions, and missing telemetry. Closure requires causal explanation, reproducibility, validated recovery, and residual limitations.

### Engineering practices

- Define authority, ownership, desired state, dependencies, limits, and evidence for source of truth and ingestion.
- Use typed APIs, stable identities, explicit state machines, idempotency, and bounded effects.
- Separate control plane, execution plane, data plane, evidence plane, and human decision authority.
- Test nominal, prohibited, overloaded, partitioned, partially failed, recovery, upgrade, and rollback cases.
- Preserve configuration, policy, model, source, trace, event, artifact, approval, and user-outcome evidence.
- Scale through quotas, queues, backpressure, fairness, isolation, canaries, and deterministic fan-in.

### Implementation sketch

```
platform_request:
  api: ApplicationEnvironment/v2
  owner: team-payments
  environment: production
  composition:
    - kubernetes-namespace
    - workload-identity
    - database
    - gateway-route
    - telemetry
  policies:
    - signed-artifacts
    - restricted-pods
    - encrypted-data
  gitops:
    repository: environments-prod
    promotion: pull-request
  evidence:
    require: [policy-tests, drift-check, rollback-plan]
```

### Failure patterns

Treating source of truth and ingestion as a feature rather than an end-to-end stateful system.

Scaling worker count or deployment scope before bounding queues, budgets, retries, and fan-in.

Trusting controller, agent, or proxy status without validating external state and user outcome.

Sharing credentials, workspaces, caches, or context across tenants and roles without explicit policy.

Allowing AI-generated plans, policies, or changes to bypass deterministic validation and human authority.

Declaring recovery after process restart without reconciling orphan effects, leases, artifacts, and evidence.

### Validation and evidence

| Method | Expected evidence |
|---|---|
| Examine | Requirements, APIs, identities, state machines, policies, versions, resources, and dependencies for source of truth and ingestion. |
| Observe | Queue, task, controller, model, tool, route, proxy, event, resource, trace, artifact, and user-visible state. |
| Test | Expected, prohibited, malformed, duplicate, overloaded, partitioned, failed, recovery, upgrade, and rollback conditions. |
| Measure | Task success, reliability, latency, throughput, fairness, cost, token and compute use, blast radius, and operator burden. |
| Reconcile | Desired and planned state against actual cloud, cluster, agent, network, data, external-effect, and evidence state. |

## 2.3 Portal and documentation

Portal and documentation must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Provide discovery, TechDocs, scorecards, dashboards, runbooks, costs, incidents, and support. The design should identify authoritative state, workload identities, typed contracts, resource and token budgets, trust boundaries, failure domains, telemetry, rollback, and acceptance evidence.

This guide traces the subject through the developer intent, catalog, template or API, policy, Git source, reconciliation, platform resource, evidence, and user outcome chain. A running agent, reconciled object, successful API call, healthy proxy, or completed function is not proof of the intended outcome. Desired state must be compared with applied state, external effects, negative tests, degraded behavior, causal evidence, cost, and user or mission results.

Production engineering begins with a minimal known-good control plane and explicit invariants. Add one worker class, model route, controller, policy, event source, gateway, or platform feature at a time; preserve before-state evidence; test authorized and prohibited behavior; inject realistic failures; and retain a tested compensation or rollback path.

Troubleshooting locates the first divergence from the expected state machine. Account for stale watches, leases, retries, eventual consistency, duplicated work, cache state, identity rotation, partial external effects, version skew, prompt injection,

overload, network partitions, and missing telemetry. Closure requires causal explanation, reproducibility, validated recovery, and residual limitations.

### Engineering practices

- Define authority, ownership, desired state, dependencies, limits, and evidence for portal and documentation.
- Use typed APIs, stable identities, explicit state machines, idempotency, and bounded effects.
- Separate control plane, execution plane, data plane, evidence plane, and human decision authority.
- Test nominal, prohibited, overloaded, partitioned, partially failed, recovery, upgrade, and rollback cases.
- Preserve configuration, policy, model, source, trace, event, artifact, approval, and user-outcome evidence.
- Scale through quotas, queues, backpressure, fairness, isolation, canaries, and deterministic fan-in.

### Failure patterns

- Treating portal and documentation as a feature rather than an end-to-end stateful system.
- Scaling worker count or deployment scope before bounding queues, budgets, retries, and fan-in.
- Trusting controller, agent, or proxy status without validating external state and user outcome.
- Sharing credentials, workspaces, caches, or context across tenants and roles without explicit policy.
- Allowing AI-generated plans, policies, or changes to bypass deterministic validation and human authority.
- Declaring recovery after process restart without reconciling orphan effects, leases, artifacts, and evidence.

### Validation and evidence

| Method | Expected evidence |
| --- | --- |
| Examine | Requirements, APIs, identities, state machines, policies, versions, resources, and dependencies for portal and documentation. |
| Observe | Queue, task, controller, model, tool, route, proxy, event, resource, trace, artifact, and user-visible state. |
| Test | Expected, prohibited, malformed, duplicate, overloaded, partitioned, failed, recovery, upgrade, and rollback conditions. |
| Measure | Task success, reliability, latency, throughput, fairness, cost, token and compute use, blast radius, and operator burden. |
| Reconcile | Desired and planned state against actual cloud, cluster, agent, network, data, external-effect, and evidence state. |

## 2.4 Ownership and dependency graph

Ownership and dependency graph must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Connect code, services, APIs, data, infrastructure, teams, SLOs, vulnerabilities, and changes. The design should identify authoritative state, workload identities, typed contracts, resource and token budgets, trust boundaries, failure domains, telemetry, rollback, and acceptance evidence.

This guide traces the subject through the developer intent, catalog, template or API, policy, Git source, reconciliation, platform resource, evidence, and user outcome chain. A running agent, reconciled object, successful API call, healthy proxy, or completed function is not proof of the intended outcome. Desired state must be compared with applied state, external effects, negative tests, degraded behavior, causal evidence, cost, and user or mission results.

Production engineering begins with a minimal known-good control plane and explicit invariants. Add one worker class, model route, controller, policy, event source, gateway, or platform feature at a time; preserve before-state evidence; test authorized and prohibited behavior; inject realistic failures; and retain a tested compensation or rollback path.

Troubleshooting locates the first divergence from the expected state machine. Account for stale watches, leases, retries, eventual consistency, duplicated work, cache state, identity rotation, partial external effects, version skew, prompt injection, overload, network partitions, and missing telemetry. Closure requires causal explanation, reproducibility, validated recovery, and residual limitations.

### Engineering practices

- Define authority, ownership, desired state, dependencies, limits, and evidence for ownership and dependency graph.

Use typed APIs, stable identities, explicit state machines, idempotency, and bounded effects.

Separate control plane, execution plane, data plane, evidence plane, and human decision authority.

Test nominal, prohibited, overloaded, partitioned, partially failed, recovery, upgrade, and rollback cases.

Preserve configuration, policy, model, source, trace, event, artifact, approval, and user-outcome evidence.

Scale through quotas, queues, backpressure, fairness, isolation, canaries, and deterministic fan-in.

## Failure patterns

Treating ownership and dependency graph as a feature rather than an end-to-end stateful system.

Scaling worker count or deployment scope before bounding queues, budgets, retries, and fan-in.

Trusting controller, agent, or proxy status without validating external state and user outcome.

Sharing credentials, workspaces, caches, or context across tenants and roles without explicit policy.

Allowing AI-generated plans, policies, or changes to bypass deterministic validation and human authority.

Declaring recovery after process restart without reconciling orphan effects, leases, artifacts, and evidence.

## Validation and evidence

| Method | Expected evidence |
| --- | --- |
| Examine | Requirements, APIs, identities, state machines, policies, versions, resources, and dependencies for ownership and dependency graph. |
| Observe | Queue, task, controller, model, tool, route, proxy, event, resource, trace, artifact, and user-visible state. |
| Test | Expected, prohibited, malformed, duplicate, overloaded, partitioned, failed, recovery, upgrade, and rollback conditions. |
| Measure | Task success, reliability, latency, throughput, fairness, cost, token and compute use, blast radius, and operator burden. |
| Reconcile | Desired and planned state against actual cloud, cluster, agent, network, data, external-effect, and evidence state. |

# Chapter 02 laboratory and review

Practical laboratory
Create a software catalog for one product and identify orphan services, undocumented APIs, and ownership conflicts.

## Laboratory evidence package

- Architecture or topology showing boundaries, identities, dependencies, and authoritative inputs.
- Versioned configuration or code with explanatory comments and reproducible setup instructions.
- Nominal-path validation plus at least two injected failure or boundary conditions.
- Structured logs, traces, metrics, packet captures, test output, or other appropriate evidence.
- A concise decision record covering findings, limitations, residual risks, and next actions.

## Frontier extension

Explore continuously generated architecture graphs from source, runtime, and cloud evidence.

Review gate
Advance only when another engineer can reproduce the result, explain the architecture, identify the failure boundary, and distinguish measured evidence from assumptions or projections.

CHAPTER 03

# Golden paths, templates, and self-service

Turn organizational knowledge into safe repeatable workflows.

Chapter operating position
Turn organizational knowledge into safe repeatable workflows.

## 3.1 Golden path design

Golden path design must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Bundle architecture, repository, build, runtime, observability, security, docs, ownership, and lifecycle. The design should identify authoritative state, workload identities, typed contracts, resource and token budgets, trust boundaries, failure domains, telemetry, rollback, and acceptance evidence.

This guide traces the subject through the developer intent, catalog, template or API, policy, Git source, reconciliation, platform resource, evidence, and user outcome chain. A running agent, reconciled object, successful API call, healthy proxy, or completed function is not proof of the intended outcome. Desired state must be compared with applied state, external effects, negative tests, degraded behavior, causal evidence, cost, and user or mission results.

Production engineering begins with a minimal known-good control plane and explicit invariants. Add one worker class, model route, controller, policy, event source, gateway, or platform feature at a time; preserve before-state evidence; test authorized and prohibited behavior; inject realistic failures; and retain a tested compensation or rollback path.

Troubleshooting locates the first divergence from the expected state machine. Account for stale watches, leases, retries, eventual consistency, duplicated work, cache state, identity rotation, partial external effects, version skew, prompt injection, overload, network partitions, and missing telemetry. Closure requires causal explanation, reproducibility, validated recovery, and residual limitations.

### Engineering practices

- Define authority, ownership, desired state, dependencies, limits, and evidence for golden path design.
- Use typed APIs, stable identities, explicit state machines, idempotency, and bounded effects.
- Separate control plane, execution plane, data plane, evidence plane, and human decision authority.
- Test nominal, prohibited, overloaded, partitioned, partially failed, recovery, upgrade, and rollback cases.
- Preserve configuration, policy, model, source, trace, event, artifact, approval, and user-outcome evidence.
- Scale through quotas, queues, backpressure, fairness, isolation, canaries, and deterministic fan-in.

### Failure patterns

- Treating golden path design as a feature rather than an end-to-end stateful system.
- Scaling worker count or deployment scope before bounding queues, budgets, retries, and fan-in.
- Trusting controller, agent, or proxy status without validating external state and user outcome.
- Sharing credentials, workspaces, caches, or context across tenants and roles without explicit policy.
- Allowing AI-generated plans, policies, or changes to bypass deterministic validation and human authority.
- Declaring recovery after process restart without reconciling orphan effects, leases, artifacts, and evidence.

### Validation and evidence

| Method | Expected evidence |
| --- | --- |
| Examine | Requirements, APIs, identities, state machines, policies, versions, resources, and dependencies for golden path design. |
| Observe | Queue, task, controller, model, tool, route, proxy, event, resource, trace, artifact, and user-visible state. |
| Test | Expected, prohibited, malformed, duplicate, overloaded, partitioned, failed, recovery, upgrade, and rollback conditions. |
| Measure | Task success, reliability, latency, throughput, fairness, cost, token and compute use, blast radius, and operator burden. |
| Reconcile | Desired and planned state against actual cloud, cluster, agent, network, data, external-effect, and evidence state. |

## 3.2 Software templates and scaffolding

Software templates and scaffolding must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Collect typed parameters, authorize actions, create repositories, generate code, and publish catalog entries. The design should identify authoritative state, workload identities, typed contracts, resource and token budgets, trust boundaries, failure domains, telemetry, rollback, and acceptance evidence.

This guide traces the subject through the developer intent, catalog, template or API, policy, Git source, reconciliation, platform resource, evidence, and user outcome chain. A running agent, reconciled object, successful API call, healthy proxy, or completed function is not proof of the intended outcome. Desired state must be compared with applied state, external effects, negative tests, degraded behavior, causal evidence, cost, and user or mission results.

Production engineering begins with a minimal known-good control plane and explicit invariants. Add one worker class, model route, controller, policy, event source, gateway, or platform feature at a time; preserve before-state evidence; test authorized and prohibited behavior; inject realistic failures; and retain a tested compensation or rollback path.

Troubleshooting locates the first divergence from the expected state machine. Account for stale watches, leases, retries, eventual consistency, duplicated work, cache state, identity rotation, partial external effects, version skew, prompt injection, overload, network partitions, and missing telemetry. Closure requires causal explanation, reproducibility, validated recovery, and residual limitations.

### Engineering practices

- Define authority, ownership, desired state, dependencies, limits, and evidence for software templates and scaffolding.
- Use typed APIs, stable identities, explicit state machines, idempotency, and bounded effects.
- Separate control plane, execution plane, data plane, evidence plane, and human decision authority.
- Test nominal, prohibited, overloaded, partitioned, partially failed, recovery, upgrade, and rollback cases.
- Preserve configuration, policy, model, source, trace, event, artifact, approval, and user-outcome evidence.
- Scale through quotas, queues, backpressure, fairness, isolation, canaries, and deterministic fan-in.

### Implementation sketch

```
platform_request:
  api: ApplicationEnvironment/v2
  owner: team-payments
  environment: production
  composition:
    - kubernetes-namespace
    - workload-identity
    - database
    - gateway-route
    - telemetry
  policies:
    - signed-artifacts
    - restricted-pods
    - encrypted-data
  gitops:
    repository: environments-prod
    promotion: pull-request
  evidence:
    require: [policy-tests, drift-check, rollback-plan]
```

### Failure patterns

- Treating software templates and scaffolding as a feature rather than an end-to-end stateful system.
- Scaling worker count or deployment scope before bounding queues, budgets, retries, and fan-in.
- Trusting controller, agent, or proxy status without validating external state and user outcome.
- Sharing credentials, workspaces, caches, or context across tenants and roles without explicit policy.
- Allowing AI-generated plans, policies, or changes to bypass deterministic validation and human authority.
- Declaring recovery after process restart without reconciling orphan effects, leases, artifacts, and evidence.

### Validation and evidence

| Method | Expected evidence |
|---|---|
| Examine | Requirements, APIs, identities, state machines, policies, versions, resources, and dependencies for software templates and scaffolding. |
| Observe | Queue, task, controller, model, tool, route, proxy, event, resource, trace, artifact, and user-visible state. |
| Test | Expected, prohibited, malformed, duplicate, overloaded, partitioned, failed, recovery, upgrade, and rollback conditions. |
| Measure | Task success, reliability, latency, throughput, fairness, cost, token and compute use, blast radius, and operator burden. |
| Reconcile | Desired and planned state against actual cloud, cluster, agent, network, data, external-effect, and evidence state. |

## 3.3 Day-two actions

Day-two actions must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Support upgrades, scaling, database changes, certificate rotation, incident operations, and decommission. The design should identify authoritative state, workload identities, typed contracts, resource and token budgets, trust boundaries, failure domains, telemetry, rollback, and acceptance evidence.

This guide traces the subject through the developer intent, catalog, template or API, policy, Git source, reconciliation, platform resource, evidence, and user outcome chain. A running agent, reconciled object, successful API call, healthy proxy, or completed function is not proof of the intended outcome. Desired state must be compared with applied state, external effects, negative tests, degraded behavior, causal evidence, cost, and user or mission results.

Production engineering begins with a minimal known-good control plane and explicit invariants. Add one worker class, model route, controller, policy, event source, gateway, or platform feature at a time; preserve before-state evidence; test authorized and prohibited behavior; inject realistic failures; and retain a tested compensation or rollback path.

Troubleshooting locates the first divergence from the expected state machine. Account for stale watches, leases, retries, eventual consistency, duplicated work, cache state, identity rotation, partial external effects, version skew, prompt injection, overload, network partitions, and missing telemetry. Closure requires causal explanation, reproducibility, validated recovery, and residual limitations.

### Engineering practices

- Define authority, ownership, desired state, dependencies, limits, and evidence for day-two actions.
- Use typed APIs, stable identities, explicit state machines, idempotency, and bounded effects.
- Separate control plane, execution plane, data plane, evidence plane, and human decision authority.
- Test nominal, prohibited, overloaded, partitioned, partially failed, recovery, upgrade, and rollback cases.
- Preserve configuration, policy, model, source, trace, event, artifact, approval, and user-outcome evidence.
- Scale through quotas, queues, backpressure, fairness, isolation, canaries, and deterministic fan-in.

### Failure patterns

- Treating day-two actions as a feature rather than an end-to-end stateful system.
- Scaling worker count or deployment scope before bounding queues, budgets, retries, and fan-in.
- Trusting controller, agent, or proxy status without validating external state and user outcome.

Sharing credentials, workspaces, caches, or context across tenants and roles without explicit policy.

Allowing AI-generated plans, policies, or changes to bypass deterministic validation and human authority.

Declaring recovery after process restart without reconciling orphan effects, leases, artifacts, and evidence.

### Validation and evidence

| Method | Expected evidence |
|---|---|
| Examine | Requirements, APIs, identities, state machines, policies, versions, resources, and dependencies for day-two actions. |
| Observe | Queue, task, controller, model, tool, route, proxy, event, resource, trace, artifact, and user-visible state. |
| Test | Expected, prohibited, malformed, duplicate, overloaded, partitioned, failed, recovery, upgrade, and rollback conditions. |
| Measure | Task success, reliability, latency, throughput, fairness, cost, token and compute use, blast radius, and operator burden. |
| Reconcile | Desired and planned state against actual cloud, cluster, agent, network, data, external-effect, and evidence state. |

## 3.4 Exceptions and escape hatches

Exceptions and escape hatches must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Provide documented alternatives, review, compensating controls, expiry, and reintegration. The design should identify authoritative state, workload identities, typed contracts, resource and token budgets, trust boundaries, failure domains, telemetry, rollback, and acceptance evidence.

This guide traces the subject through the developer intent, catalog, template or API, policy, Git source, reconciliation, platform resource, evidence, and user outcome chain. A running agent, reconciled object, successful API call, healthy proxy, or completed function is not proof of the intended outcome. Desired state must be compared with applied state, external effects, negative tests, degraded behavior, causal evidence, cost, and user or mission results.

Production engineering begins with a minimal known-good control plane and explicit invariants. Add one worker class, model route, controller, policy, event source, gateway, or platform feature at a time; preserve before-state evidence; test authorized and prohibited behavior; inject realistic failures; and retain a tested compensation or rollback path.

Troubleshooting locates the first divergence from the expected state machine. Account for stale watches, leases, retries, eventual consistency, duplicated work, cache state, identity rotation, partial external effects, version skew, prompt injection, overload, network partitions, and missing telemetry. Closure requires causal explanation, reproducibility, validated recovery, and residual limitations.

### Engineering practices

Define authority, ownership, desired state, dependencies, limits, and evidence for exceptions and escape hatches.

Use typed APIs, stable identities, explicit state machines, idempotency, and bounded effects.

Separate control plane, execution plane, data plane, evidence plane, and human decision authority.

Test nominal, prohibited, overloaded, partitioned, partially failed, recovery, upgrade, and rollback cases.

Preserve configuration, policy, model, source, trace, event, artifact, approval, and user-outcome evidence.

Scale through quotas, queues, backpressure, fairness, isolation, canaries, and deterministic fan-in.

### Failure patterns

Treating exceptions and escape hatches as a feature rather than an end-to-end stateful system.

Scaling worker count or deployment scope before bounding queues, budgets, retries, and fan-in.

Trusting controller, agent, or proxy status without validating external state and user outcome.

Sharing credentials, workspaces, caches, or context across tenants and roles without explicit policy.

Allowing AI-generated plans, policies, or changes to bypass deterministic validation and human authority.

Declaring recovery after process restart without reconciling orphan effects, leases, artifacts, and evidence.

## Validation and evidence

| Method | Expected evidence |
| --- | --- |
| Examine | Requirements, APIs, identities, state machines, policies, versions, resources, and dependencies for exceptions and escape hatches. |
| Observe | Queue, task, controller, model, tool, route, proxy, event, resource, trace, artifact, and user-visible state. |
| Test | Expected, prohibited, malformed, duplicate, overloaded, partitioned, failed, recovery, upgrade, and rollback conditions. |
| Measure | Task success, reliability, latency, throughput, fairness, cost, token and compute use, blast radius, and operator burden. |
| Reconcile | Desired and planned state against actual cloud, cluster, agent, network, data, external-effect, and evidence state. |

# Chapter 03 laboratory and review

Practical laboratory
Build a golden path for a service including repository, CI, deployment, policy, telemetry, catalog, and deletion.

## Laboratory evidence package

- Architecture or topology showing boundaries, identities, dependencies, and authoritative inputs.
- Versioned configuration or code with explanatory comments and reproducible setup instructions.
- Nominal-path validation plus at least two injected failure or boundary conditions.
- Structured logs, traces, metrics, packet captures, test output, or other appropriate evidence.
- A concise decision record covering findings, limitations, residual risks, and next actions.

## Frontier extension

Explore agent-generated golden paths whose outputs remain policy-verified and owner-approved.

Review gate
Advance only when another engineer can reproduce the result, explain the architecture, identify the failure boundary, and distinguish measured evidence from assumptions or projections.

CHAPTER 04

# Control planes and platform APIs

Expose desired state instead of implementation-specific ticket queues.

Chapter operating position
Expose desired state instead of implementation-specific ticket queues.

## 4.1 Platform API design

Platform API design must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Define domain resources, schemas, defaults, composition, status, conditions, and lifecycle. The design should identify authoritative state, workload identities, typed contracts, resource and token budgets, trust boundaries, failure domains, telemetry, rollback, and acceptance evidence.

This guide traces the subject through the developer intent, catalog, template or API, policy, Git source, reconciliation, platform resource, evidence, and user outcome chain. A running agent, reconciled object, successful API call, healthy proxy, or completed function is not proof of the intended outcome. Desired state must be compared with applied state, external effects, negative tests, degraded behavior, causal evidence, cost, and user or mission results.

Production engineering begins with a minimal known-good control plane and explicit invariants. Add one worker class, model route, controller, policy, event source, gateway, or platform feature at a time; preserve before-state evidence; test authorized and prohibited behavior; inject realistic failures; and retain a tested compensation or rollback path.

Troubleshooting locates the first divergence from the expected state machine. Account for stale watches, leases, retries, eventual consistency, duplicated work, cache state, identity rotation, partial external effects, version skew, prompt injection, overload, network partitions, and missing telemetry. Closure requires causal explanation, reproducibility, validated recovery, and residual limitations.

### Engineering practices

- Define authority, ownership, desired state, dependencies, limits, and evidence for platform api design.
- Use typed APIs, stable identities, explicit state machines, idempotency, and bounded effects.
- Separate control plane, execution plane, data plane, evidence plane, and human decision authority.
- Test nominal, prohibited, overloaded, partitioned, partially failed, recovery, upgrade, and rollback cases.
- Preserve configuration, policy, model, source, trace, event, artifact, approval, and user-outcome evidence.
- Scale through quotas, queues, backpressure, fairness, isolation, canaries, and deterministic fan-in.

### Failure patterns

- Treating platform api design as a feature rather than an end-to-end stateful system.
- Scaling worker count or deployment scope before bounding queues, budgets, retries, and fan-in.
- Trusting controller, agent, or proxy status without validating external state and user outcome.
- Sharing credentials, workspaces, caches, or context across tenants and roles without explicit policy.
- Allowing AI-generated plans, policies, or changes to bypass deterministic validation and human authority.
- Declaring recovery after process restart without reconciling orphan effects, leases, artifacts, and evidence.

### Validation and evidence

| Method | Expected evidence |
| --- | --- |
| Examine | Requirements, APIs, identities, state machines, policies, versions, resources, and dependencies for platform api design. |
| Observe | Queue, task, controller, model, tool, route, proxy, event, resource, trace, artifact, and user-visible state. |
| Test | Expected, prohibited, malformed, duplicate, overloaded, partitioned, failed, recovery, upgrade, and rollback conditions. |
| Measure | Task success, reliability, latency, throughput, fairness, cost, token and compute use, blast radius, and operator burden. |
| Reconcile | Desired and planned state against actual cloud, cluster, agent, network, data, external-effect, and evidence state. |

## 4.2 Crossplane composition

Crossplane composition must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Combine managed resources, Kubernetes resources, functions, packages, and operational workflows. The design should identify authoritative state, workload identities, typed contracts, resource and token budgets, trust boundaries, failure domains, telemetry, rollback, and acceptance evidence.

This guide traces the subject through the developer intent, catalog, template or API, policy, Git source, reconciliation, platform resource, evidence, and user outcome chain. A running agent, reconciled object, successful API call, healthy proxy, or completed function is not proof of the intended outcome. Desired state must be compared with applied state, external effects, negative tests, degraded behavior, causal evidence, cost, and user or mission results.

Production engineering begins with a minimal known-good control plane and explicit invariants. Add one worker class, model route, controller, policy, event source, gateway, or platform feature at a time; preserve before-state evidence; test authorized and prohibited behavior; inject realistic failures; and retain a tested compensation or rollback path.

Troubleshooting locates the first divergence from the expected state machine. Account for stale watches, leases, retries, eventual consistency, duplicated work, cache state, identity rotation, partial external effects, version skew, prompt injection, overload, network partitions, and missing telemetry. Closure requires causal explanation, reproducibility, validated recovery, and residual limitations.

### Engineering practices

- Define authority, ownership, desired state, dependencies, limits, and evidence for crossplane composition.
- Use typed APIs, stable identities, explicit state machines, idempotency, and bounded effects.
- Separate control plane, execution plane, data plane, evidence plane, and human decision authority.
- Test nominal, prohibited, overloaded, partitioned, partially failed, recovery, upgrade, and rollback cases.
- Preserve configuration, policy, model, source, trace, event, artifact, approval, and user-outcome evidence.
- Scale through quotas, queues, backpressure, fairness, isolation, canaries, and deterministic fan-in.

### Implementation sketch

```
platform_request:
  api: ApplicationEnvironment/v2
  owner: team-payments
  environment: production
  composition:
    - kubernetes-namespace
    - workload-identity
    - database
    - gateway-route
    - telemetry
  policies:
    - signed-artifacts
    - restricted-pods
    - encrypted-data
  gitops:
    repository: environments-prod
    promotion: pull-request
  evidence:
    require: [policy-tests, drift-check, rollback-plan]
```

### Failure patterns

- Treating crossplane composition as a feature rather than an end-to-end stateful system.
- Scaling worker count or deployment scope before bounding queues, budgets, retries, and fan-in.
- Trusting controller, agent, or proxy status without validating external state and user outcome.
- Sharing credentials, workspaces, caches, or context across tenants and roles without explicit policy.
- Allowing AI-generated plans, policies, or changes to bypass deterministic validation and human authority.
- Declaring recovery after process restart without reconciling orphan effects, leases, artifacts, and evidence.

### Validation and evidence

| Method | Expected evidence |
|---|---|
| Examine | Requirements, APIs, identities, state machines, policies, versions, resources, and dependencies for crossplane composition. |
| Observe | Queue, task, controller, model, tool, route, proxy, event, resource, trace, artifact, and user-visible state. |
| Test | Expected, prohibited, malformed, duplicate, overloaded, partitioned, failed, recovery, upgrade, and rollback conditions. |
| Measure | Task success, reliability, latency, throughput, fairness, cost, token and compute use, blast radius, and operator burden. |
| Reconcile | Desired and planned state against actual cloud, cluster, agent, network, data, external-effect, and evidence state. |

## 4.3 Reconciliation and drift

Reconciliation and drift must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Continuously compare desired, composed, external, and observed state with safe correction. The design should identify authoritative state, workload identities, typed contracts, resource and token budgets, trust boundaries, failure domains, telemetry, rollback, and acceptance evidence.

This guide traces the subject through the developer intent, catalog, template or API, policy, Git source, reconciliation, platform resource, evidence, and user outcome chain. A running agent, reconciled object, successful API call, healthy proxy, or completed function is not proof of the intended outcome. Desired state must be compared with applied state, external effects, negative tests, degraded behavior, causal evidence, cost, and user or mission results.

Production engineering begins with a minimal known-good control plane and explicit invariants. Add one worker class, model route, controller, policy, event source, gateway, or platform feature at a time; preserve before-state evidence; test authorized and prohibited behavior; inject realistic failures; and retain a tested compensation or rollback path.

Troubleshooting locates the first divergence from the expected state machine. Account for stale watches, leases, retries, eventual consistency, duplicated work, cache state, identity rotation, partial external effects, version skew, prompt injection, overload, network partitions, and missing telemetry. Closure requires causal explanation, reproducibility, validated recovery, and residual limitations.

### Engineering practices

- Define authority, ownership, desired state, dependencies, limits, and evidence for reconciliation and drift.
- Use typed APIs, stable identities, explicit state machines, idempotency, and bounded effects.
- Separate control plane, execution plane, data plane, evidence plane, and human decision authority.
- Test nominal, prohibited, overloaded, partitioned, partially failed, recovery, upgrade, and rollback cases.
- Preserve configuration, policy, model, source, trace, event, artifact, approval, and user-outcome evidence.
- Scale through quotas, queues, backpressure, fairness, isolation, canaries, and deterministic fan-in.

### Failure patterns

- Treating reconciliation and drift as a feature rather than an end-to-end stateful system.
- Scaling worker count or deployment scope before bounding queues, budgets, retries, and fan-in.
- Trusting controller, agent, or proxy status without validating external state and user outcome.

Sharing credentials, workspaces, caches, or context across tenants and roles without explicit policy.

Allowing AI-generated plans, policies, or changes to bypass deterministic validation and human authority.

Declaring recovery after process restart without reconciling orphan effects, leases, artifacts, and evidence.

### Validation and evidence

| Method | Expected evidence |
|---|---|
| Examine | Requirements, APIs, identities, state machines, policies, versions, resources, and dependencies for reconciliation and drift. |
| Observe | Queue, task, controller, model, tool, route, proxy, event, resource, trace, artifact, and user-visible state. |
| Test | Expected, prohibited, malformed, duplicate, overloaded, partitioned, failed, recovery, upgrade, and rollback conditions. |
| Measure | Task success, reliability, latency, throughput, fairness, cost, token and compute use, blast radius, and operator burden. |
| Reconcile | Desired and planned state against actual cloud, cluster, agent, network, data, external-effect, and evidence state. |

## 4.4 Versioning and evolution

Versioning and evolution must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Migrate APIs, compositions, providers, resources, and consumers without uncontrolled breakage. The design should identify authoritative state, workload identities, typed contracts, resource and token budgets, trust boundaries, failure domains, telemetry, rollback, and acceptance evidence.

This guide traces the subject through the developer intent, catalog, template or API, policy, Git source, reconciliation, platform resource, evidence, and user outcome chain. A running agent, reconciled object, successful API call, healthy proxy, or completed function is not proof of the intended outcome. Desired state must be compared with applied state, external effects, negative tests, degraded behavior, causal evidence, cost, and user or mission results.

Production engineering begins with a minimal known-good control plane and explicit invariants. Add one worker class, model route, controller, policy, event source, gateway, or platform feature at a time; preserve before-state evidence; test authorized and prohibited behavior; inject realistic failures; and retain a tested compensation or rollback path.

Troubleshooting locates the first divergence from the expected state machine. Account for stale watches, leases, retries, eventual consistency, duplicated work, cache state, identity rotation, partial external effects, version skew, prompt injection, overload, network partitions, and missing telemetry. Closure requires causal explanation, reproducibility, validated recovery, and residual limitations.

### Engineering practices

Define authority, ownership, desired state, dependencies, limits, and evidence for versioning and evolution.

Use typed APIs, stable identities, explicit state machines, idempotency, and bounded effects.

Separate control plane, execution plane, data plane, evidence plane, and human decision authority.

Test nominal, prohibited, overloaded, partitioned, partially failed, recovery, upgrade, and rollback cases.

Preserve configuration, policy, model, source, trace, event, artifact, approval, and user-outcome evidence.

Scale through quotas, queues, backpressure, fairness, isolation, canaries, and deterministic fan-in.

### Failure patterns

Treating versioning and evolution as a feature rather than an end-to-end stateful system.

Scaling worker count or deployment scope before bounding queues, budgets, retries, and fan-in.

Trusting controller, agent, or proxy status without validating external state and user outcome.

Sharing credentials, workspaces, caches, or context across tenants and roles without explicit policy.

Allowing AI-generated plans, policies, or changes to bypass deterministic validation and human authority.

Declaring recovery after process restart without reconciling orphan effects, leases, artifacts, and evidence.

## Validation and evidence

| Method | Expected evidence |
| --- | --- |
| Examine | Requirements, APIs, identities, state machines, policies, versions, resources, and dependencies for versioning and evolution. |
| Observe | Queue, task, controller, model, tool, route, proxy, event, resource, trace, artifact, and user-visible state. |
| Test | Expected, prohibited, malformed, duplicate, overloaded, partitioned, failed, recovery, upgrade, and rollback conditions. |
| Measure | Task success, reliability, latency, throughput, fairness, cost, token and compute use, blast radius, and operator burden. |
| Reconcile | Desired and planned state against actual cloud, cluster, agent, network, data, external-effect, and evidence state. |

# Chapter 04 laboratory and review

Practical laboratory

Design a platform API for an application environment and compose cloud, Kubernetes, data, identity, and policy resources.

## Laboratory evidence package

- Architecture or topology showing boundaries, identities, dependencies, and authoritative inputs.
- Versioned configuration or code with explanatory comments and reproducible setup instructions.
- Nominal-path validation plus at least two injected failure or boundary conditions.
- Structured logs, traces, metrics, packet captures, test output, or other appropriate evidence.
- A concise decision record covering findings, limitations, residual risks, and next actions.

## Frontier extension

Explore multi-control-plane fabrics spanning applications, agents, cloud, data, and sovereign infrastructure.

Review gate

Advance only when another engineer can reproduce the result, explain the architecture, identify the failure boundary, and distinguish measured evidence from assumptions or projections.

CHAPTER 05

# GitOps architecture and promotion

Use declarative sources and reconciliation as the delivery control system.

Chapter operating position
Use declarative sources and reconciliation as the delivery control system.

## 5.1 Source authority and repository structure

Source authority and repository structure must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Define desired-state repositories, ownership, branches, environments, generators, secrets, and dependencies. The design should identify authoritative state, workload identities, typed contracts, resource and token budgets, trust boundaries, failure domains, telemetry, rollback, and acceptance evidence.

This guide traces the subject through the developer intent, catalog, template or API, policy, Git source, reconciliation, platform resource, evidence, and user outcome chain. A running agent, reconciled object, successful API call, healthy proxy, or completed function is not proof of the intended outcome. Desired state must be compared with applied state, external effects, negative tests, degraded behavior, causal evidence, cost, and user or mission results.

Production engineering begins with a minimal known-good control plane and explicit invariants. Add one worker class, model route, controller, policy, event source, gateway, or platform feature at a time; preserve before-state evidence; test authorized and prohibited behavior; inject realistic failures; and retain a tested compensation or rollback path.

Troubleshooting locates the first divergence from the expected state machine. Account for stale watches, leases, retries, eventual consistency, duplicated work, cache state, identity rotation, partial external effects, version skew, prompt injection, overload, network partitions, and missing telemetry. Closure requires causal explanation, reproducibility, validated recovery, and residual limitations.

### Engineering practices

- Define authority, ownership, desired state, dependencies, limits, and evidence for source authority and repository structure.
- Use typed APIs, stable identities, explicit state machines, idempotency, and bounded effects.
- Separate control plane, execution plane, data plane, evidence plane, and human decision authority.
- Test nominal, prohibited, overloaded, partitioned, partially failed, recovery, upgrade, and rollback cases.
- Preserve configuration, policy, model, source, trace, event, artifact, approval, and user-outcome evidence.
- Scale through quotas, queues, backpressure, fairness, isolation, canaries, and deterministic fan-in.

### Failure patterns

- Treating source authority and repository structure as a feature rather than an end-to-end stateful system.
- Scaling worker count or deployment scope before bounding queues, budgets, retries, and fan-in.
- Trusting controller, agent, or proxy status without validating external state and user outcome.
- Sharing credentials, workspaces, caches, or context across tenants and roles without explicit policy.
- Allowing AI-generated plans, policies, or changes to bypass deterministic validation and human authority.
- Declaring recovery after process restart without reconciling orphan effects, leases, artifacts, and evidence.

### Validation and evidence

| Method | Expected evidence |
| --- | --- |
| Examine | Requirements, APIs, identities, state machines, policies, versions, resources, and dependencies for source authority and repository structure. |
| Observe | Queue, task, controller, model, tool, route, proxy, event, resource, trace, artifact, and user-visible state. |
| Test | Expected, prohibited, malformed, duplicate, overloaded, partitioned, failed, recovery, upgrade, and rollback conditions. |
| Measure | Task success, reliability, latency, throughput, fairness, cost, token and compute use, blast radius, and operator burden. |
| Reconcile | Desired and planned state against actual cloud, cluster, agent, network, data, external-effect, and evidence state. |

## 5.2 Reconciliation loops

Reconciliation loops must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Fetch, render, compare, apply, health-check, retry, suspend, prune, and report drift. The design should identify authoritative state, workload identities, typed contracts, resource and token budgets, trust boundaries, failure domains, telemetry, rollback, and acceptance evidence.

This guide traces the subject through the developer intent, catalog, template or API, policy, Git source, reconciliation, platform resource, evidence, and user outcome chain. A running agent, reconciled object, successful API call, healthy proxy, or completed function is not proof of the intended outcome. Desired state must be compared with applied state, external effects, negative tests, degraded behavior, causal evidence, cost, and user or mission results.

Production engineering begins with a minimal known-good control plane and explicit invariants. Add one worker class, model route, controller, policy, event source, gateway, or platform feature at a time; preserve before-state evidence; test authorized and prohibited behavior; inject realistic failures; and retain a tested compensation or rollback path.

Troubleshooting locates the first divergence from the expected state machine. Account for stale watches, leases, retries, eventual consistency, duplicated work, cache state, identity rotation, partial external effects, version skew, prompt injection, overload, network partitions, and missing telemetry. Closure requires causal explanation, reproducibility, validated recovery, and residual limitations.

### Engineering practices

- Define authority, ownership, desired state, dependencies, limits, and evidence for reconciliation loops.
- Use typed APIs, stable identities, explicit state machines, idempotency, and bounded effects.
- Separate control plane, execution plane, data plane, evidence plane, and human decision authority.
- Test nominal, prohibited, overloaded, partitioned, partially failed, recovery, upgrade, and rollback cases.
- Preserve configuration, policy, model, source, trace, event, artifact, approval, and user-outcome evidence.
- Scale through quotas, queues, backpressure, fairness, isolation, canaries, and deterministic fan-in.

### Implementation sketch

```
platform_request:
  api: ApplicationEnvironment/v2
  owner: team-payments
  environment: production
  composition:
    - kubernetes-namespace
    - workload-identity
    - database
    - gateway-route
    - telemetry
  policies:
    - signed-artifacts
    - restricted-pods
    - encrypted-data
  gitops:
    repository: environments-prod
    promotion: pull-request
  evidence:
    require: [policy-tests, drift-check, rollback-plan]
```

### Failure patterns

- Treating reconciliation loops as a feature rather than an end-to-end stateful system.
- Scaling worker count or deployment scope before bounding queues, budgets, retries, and fan-in.
- Trusting controller, agent, or proxy status without validating external state and user outcome.
- Sharing credentials, workspaces, caches, or context across tenants and roles without explicit policy.
- Allowing AI-generated plans, policies, or changes to bypass deterministic validation and human authority.
- Declaring recovery after process restart without reconciling orphan effects, leases, artifacts, and evidence.

### Validation and evidence

| Method | Expected evidence |
| --- | --- |
| Examine | Requirements, APIs, identities, state machines, policies, versions, resources, and dependencies for reconciliation loops. |
| Observe | Queue, task, controller, model, tool, route, proxy, event, resource, trace, artifact, and user-visible state. |
| Test | Expected, prohibited, malformed, duplicate, overloaded, partitioned, failed, recovery, upgrade, and rollback conditions. |
| Measure | Task success, reliability, latency, throughput, fairness, cost, token and compute use, blast radius, and operator burden. |
| Reconcile | Desired and planned state against actual cloud, cluster, agent, network, data, external-effect, and evidence state. |

## 5.3 Promotion and release

Promotion and release must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Move immutable artifacts and configuration through environments with approvals, tests, and provenance. The design should identify authoritative state, workload identities, typed contracts, resource and token budgets, trust boundaries, failure domains, telemetry, rollback, and acceptance evidence.

This guide traces the subject through the developer intent, catalog, template or API, policy, Git source, reconciliation, platform resource, evidence, and user outcome chain. A running agent, reconciled object, successful API call, healthy proxy, or completed function is not proof of the intended outcome. Desired state must be compared with applied state, external effects, negative tests, degraded behavior, causal evidence, cost, and user or mission results.

Production engineering begins with a minimal known-good control plane and explicit invariants. Add one worker class, model route, controller, policy, event source, gateway, or platform feature at a time; preserve before-state evidence; test authorized and prohibited behavior; inject realistic failures; and retain a tested compensation or rollback path.

Troubleshooting locates the first divergence from the expected state machine. Account for stale watches, leases, retries, eventual consistency, duplicated work, cache state, identity rotation, partial external effects, version skew, prompt injection, overload, network partitions, and missing telemetry. Closure requires causal explanation, reproducibility, validated recovery, and residual limitations.

### Engineering practices

- Define authority, ownership, desired state, dependencies, limits, and evidence for promotion and release.
- Use typed APIs, stable identities, explicit state machines, idempotency, and bounded effects.
- Separate control plane, execution plane, data plane, evidence plane, and human decision authority.
- Test nominal, prohibited, overloaded, partitioned, partially failed, recovery, upgrade, and rollback cases.
- Preserve configuration, policy, model, source, trace, event, artifact, approval, and user-outcome evidence.
- Scale through quotas, queues, backpressure, fairness, isolation, canaries, and deterministic fan-in.

### Failure patterns

- Treating promotion and release as a feature rather than an end-to-end stateful system.
- Scaling worker count or deployment scope before bounding queues, budgets, retries, and fan-in.
- Trusting controller, agent, or proxy status without validating external state and user outcome.
- Sharing credentials, workspaces, caches, or context across tenants and roles without explicit policy.
- Allowing AI-generated plans, policies, or changes to bypass deterministic validation and human authority.
- Declaring recovery after process restart without reconciling orphan effects, leases, artifacts, and evidence.

### Validation and evidence

| Method | Expected evidence |
| --- | --- |
| Examine | Requirements, APIs, identities, state machines, policies, versions, resources, and dependencies for promotion and release. |
| Observe | Queue, task, controller, model, tool, route, proxy, event, resource, trace, artifact, and user-visible state. |
| Test | Expected, prohibited, malformed, duplicate, overloaded, partitioned, failed, recovery, upgrade, and rollback conditions. |
| Measure | Task success, reliability, latency, throughput, fairness, cost, token and compute use, blast radius, and operator burden. |
| Reconcile | Desired and planned state against actual cloud, cluster, agent, network, data, external-effect, and evidence state. |

## 5.4 Rollback and disaster recovery

Rollback and disaster recovery must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Revert source, restore controllers, recover clusters, reconcile external state, and validate service. The design should identify authoritative state, workload identities, typed contracts, resource and token budgets, trust boundaries, failure domains, telemetry, rollback, and acceptance evidence.

This guide traces the subject through the developer intent, catalog, template or API, policy, Git source, reconciliation, platform resource, evidence, and user outcome chain. A running agent, reconciled object, successful API call, healthy proxy, or completed function is not proof of the intended outcome. Desired state must be compared with applied state, external effects, negative tests, degraded behavior, causal evidence, cost, and user or mission results.

Production engineering begins with a minimal known-good control plane and explicit invariants. Add one worker class, model route, controller, policy, event source, gateway, or platform feature at a time; preserve before-state evidence; test authorized and prohibited behavior; inject realistic failures; and retain a tested compensation or rollback path.

Troubleshooting locates the first divergence from the expected state machine. Account for stale watches, leases, retries, eventual consistency, duplicated work, cache state, identity rotation, partial external effects, version skew, prompt injection, overload, network partitions, and missing telemetry. Closure requires causal explanation, reproducibility, validated recovery, and residual limitations.

### Engineering practices

- Define authority, ownership, desired state, dependencies, limits, and evidence for rollback and disaster recovery.
- Use typed APIs, stable identities, explicit state machines, idempotency, and bounded effects.
- Separate control plane, execution plane, data plane, evidence plane, and human decision authority.

Test nominal, prohibited, overloaded, partitioned, partially failed, recovery, upgrade, and rollback cases.

Preserve configuration, policy, model, source, trace, event, artifact, approval, and user-outcome evidence.

Scale through quotas, queues, backpressure, fairness, isolation, canaries, and deterministic fan-in.

## Failure patterns

Treating rollback and disaster recovery as a feature rather than an end-to-end stateful system.

Scaling worker count or deployment scope before bounding queues, budgets, retries, and fan-in.

Trusting controller, agent, or proxy status without validating external state and user outcome.

Sharing credentials, workspaces, caches, or context across tenants and roles without explicit policy.

Allowing AI-generated plans, policies, or changes to bypass deterministic validation and human authority.

Declaring recovery after process restart without reconciling orphan effects, leases, artifacts, and evidence.

## Validation and evidence

| Method | Expected evidence |
| --- | --- |
| Examine | Requirements, APIs, identities, state machines, policies, versions, resources, and dependencies for rollback and disaster recovery. |
| Observe | Queue, task, controller, model, tool, route, proxy, event, resource, trace, artifact, and user-visible state. |
| Test | Expected, prohibited, malformed, duplicate, overloaded, partitioned, failed, recovery, upgrade, and rollback conditions. |
| Measure | Task success, reliability, latency, throughput, fairness, cost, token and compute use, blast radius, and operator burden. |
| Reconcile | Desired and planned state against actual cloud, cluster, agent, network, data, external-effect, and evidence state. |

# Chapter 05 laboratory and review

Practical laboratory
Implement or simulate a GitOps promotion from development to canary and production with drift and rollback.

## Laboratory evidence package

- Architecture or topology showing boundaries, identities, dependencies, and authoritative inputs.
- Versioned configuration or code with explanatory comments and reproducible setup instructions.
- Nominal-path validation plus at least two injected failure or boundary conditions.
- Structured logs, traces, metrics, packet captures, test output, or other appropriate evidence.
- A concise decision record covering findings, limitations, residual risks, and next actions.

## Frontier extension

Explore transactional multi-cluster GitOps and cryptographically witnessed reconciliation.

Review gate
Advance only when another engineer can reproduce the result, explain the architecture, identify the failure boundary, and distinguish measured evidence from assumptions or projections.

CHAPTER 06

# Policy as code and enforcement architecture

Express organizational constraints as versioned, testable decisions.

Chapter operating position
Express organizational constraints as versioned, testable decisions.

## 6.1 Policy domains and decision points

Policy domains and decision points must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Apply policy in source, pull request, CI, artifact, deployment, admission, runtime, API, and data access. The design should identify authoritative state, workload identities, typed contracts, resource and token budgets, trust boundaries, failure domains, telemetry, rollback, and acceptance evidence.

This guide traces the subject through the developer intent, catalog, template or API, policy, Git source, reconciliation, platform resource, evidence, and user outcome chain. A running agent, reconciled object, successful API call, healthy proxy, or completed function is not proof of the intended outcome. Desired state must be compared with applied state, external effects, negative tests, degraded behavior, causal evidence, cost, and user or mission results.

Production engineering begins with a minimal known-good control plane and explicit invariants. Add one worker class, model route, controller, policy, event source, gateway, or platform feature at a time; preserve before-state evidence; test authorized and prohibited behavior; inject realistic failures; and retain a tested compensation or rollback path.

Troubleshooting locates the first divergence from the expected state machine. Account for stale watches, leases, retries, eventual consistency, duplicated work, cache state, identity rotation, partial external effects, version skew, prompt injection, overload, network partitions, and missing telemetry. Closure requires causal explanation, reproducibility, validated recovery, and residual limitations.

### Engineering practices

- Define authority, ownership, desired state, dependencies, limits, and evidence for policy domains and decision points.
- Use typed APIs, stable identities, explicit state machines, idempotency, and bounded effects.
- Separate control plane, execution plane, data plane, evidence plane, and human decision authority.
- Test nominal, prohibited, overloaded, partitioned, partially failed, recovery, upgrade, and rollback cases.
- Preserve configuration, policy, model, source, trace, event, artifact, approval, and user-outcome evidence.
- Scale through quotas, queues, backpressure, fairness, isolation, canaries, and deterministic fan-in.

### Failure patterns

- Treating policy domains and decision points as a feature rather than an end-to-end stateful system.
- Scaling worker count or deployment scope before bounding queues, budgets, retries, and fan-in.
- Trusting controller, agent, or proxy status without validating external state and user outcome.
- Sharing credentials, workspaces, caches, or context across tenants and roles without explicit policy.
- Allowing AI-generated plans, policies, or changes to bypass deterministic validation and human authority.
- Declaring recovery after process restart without reconciling orphan effects, leases, artifacts, and evidence.

### Validation and evidence

| Method | Expected evidence |
| --- | --- |
| Examine | Requirements, APIs, identities, state machines, policies, versions, resources, and dependencies for policy domains and decision points. |
| Observe | Queue, task, controller, model, tool, route, proxy, event, resource, trace, artifact, and user-visible state. |
| Test | Expected, prohibited, malformed, duplicate, overloaded, partitioned, failed, recovery, upgrade, and rollback conditions. |
| Measure | Task success, reliability, latency, throughput, fairness, cost, token and compute use, blast radius, and operator burden. |
| Reconcile | Desired and planned state against actual cloud, cluster, agent, network, data, external-effect, and evidence state. |

## 6.2 OPA and Rego

OPA and Rego must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Separate policy decision from enforcement, use structured input, bundles, tests, partial evaluation, and decision logs. The design should identify authoritative state, workload identities, typed contracts, resource and token budgets, trust boundaries, failure domains, telemetry, rollback, and acceptance evidence.

This guide traces the subject through the developer intent, catalog, template or API, policy, Git source, reconciliation, platform resource, evidence, and user outcome chain. A running agent, reconciled object, successful API call, healthy proxy, or completed function is not proof of the intended outcome. Desired state must be compared with applied state, external effects, negative tests, degraded behavior, causal evidence, cost, and user or mission results.

Production engineering begins with a minimal known-good control plane and explicit invariants. Add one worker class, model route, controller, policy, event source, gateway, or platform feature at a time; preserve before-state evidence; test authorized and prohibited behavior; inject realistic failures; and retain a tested compensation or rollback path.

Troubleshooting locates the first divergence from the expected state machine. Account for stale watches, leases, retries, eventual consistency, duplicated work, cache state, identity rotation, partial external effects, version skew, prompt injection, overload, network partitions, and missing telemetry. Closure requires causal explanation, reproducibility, validated recovery, and residual limitations.

### Engineering practices

- Define authority, ownership, desired state, dependencies, limits, and evidence for opa and rego.
- Use typed APIs, stable identities, explicit state machines, idempotency, and bounded effects.
- Separate control plane, execution plane, data plane, evidence plane, and human decision authority.
- Test nominal, prohibited, overloaded, partitioned, partially failed, recovery, upgrade, and rollback cases.
- Preserve configuration, policy, model, source, trace, event, artifact, approval, and user-outcome evidence.
- Scale through quotas, queues, backpressure, fairness, isolation, canaries, and deterministic fan-in.

### Implementation sketch

```
platform_request:
  api: ApplicationEnvironment/v2
  owner: team-payments
  environment: production
  composition:
    - kubernetes-namespace
    - workload-identity
    - database
    - gateway-route
    - telemetry
  policies:
    - signed-artifacts
    - restricted-pods
    - encrypted-data
  gitops:
    repository: environments-prod
    promotion: pull-request
  evidence:
    require: [policy-tests, drift-check, rollback-plan]
```

### Failure patterns

- Treating opa and rego as a feature rather than an end-to-end stateful system.
- Scaling worker count or deployment scope before bounding queues, budgets, retries, and fan-in.
- Trusting controller, agent, or proxy status without validating external state and user outcome.
- Sharing credentials, workspaces, caches, or context across tenants and roles without explicit policy.
- Allowing AI-generated plans, policies, or changes to bypass deterministic validation and human authority.
- Declaring recovery after process restart without reconciling orphan effects, leases, artifacts, and evidence.

### Validation and evidence

| Method | Expected evidence |
| --- | --- |
| Examine | Requirements, APIs, identities, state machines, policies, versions, resources, and dependencies for opa and rego. |
| Observe | Queue, task, controller, model, tool, route, proxy, event, resource, trace, artifact, and user-visible state. |
| Test | Expected, prohibited, malformed, duplicate, overloaded, partitioned, failed, recovery, upgrade, and rollback conditions. |
| Measure | Task success, reliability, latency, throughput, fairness, cost, token and compute use, blast radius, and operator burden. |
| Reconcile | Desired and planned state against actual cloud, cluster, agent, network, data, external-effect, and evidence state. |

## 6.3 Kubernetes-native and CEL policy

Kubernetes-native and CEL policy must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Use Kyverno, admission policies, mutation, validation, generation, image verification, and exceptions. The design should identify authoritative state, workload identities, typed contracts, resource and token budgets, trust boundaries, failure domains, telemetry, rollback, and acceptance evidence.

This guide traces the subject through the developer intent, catalog, template or API, policy, Git source, reconciliation, platform resource, evidence, and user outcome chain. A running agent, reconciled object, successful API call, healthy proxy, or completed function is not proof of the intended outcome. Desired state must be compared with applied state, external effects, negative tests, degraded behavior, causal evidence, cost, and user or mission results.

Production engineering begins with a minimal known-good control plane and explicit invariants. Add one worker class, model route, controller, policy, event source, gateway, or platform feature at a time; preserve before-state evidence; test authorized and prohibited behavior; inject realistic failures; and retain a tested compensation or rollback path.

Troubleshooting locates the first divergence from the expected state machine. Account for stale watches, leases, retries, eventual consistency, duplicated work, cache state, identity rotation, partial external effects, version skew, prompt injection, overload, network partitions, and missing telemetry. Closure requires causal explanation, reproducibility, validated recovery, and residual limitations.

### Engineering practices

- Define authority, ownership, desired state, dependencies, limits, and evidence for kubernetes-native and cel policy.
- Use typed APIs, stable identities, explicit state machines, idempotency, and bounded effects.
- Separate control plane, execution plane, data plane, evidence plane, and human decision authority.
- Test nominal, prohibited, overloaded, partitioned, partially failed, recovery, upgrade, and rollback cases.
- Preserve configuration, policy, model, source, trace, event, artifact, approval, and user-outcome evidence.
- Scale through quotas, queues, backpressure, fairness, isolation, canaries, and deterministic fan-in.

### Failure patterns

- Treating kubernetes-native and cel policy as a feature rather than an end-to-end stateful system.
- Scaling worker count or deployment scope before bounding queues, budgets, retries, and fan-in.
- Trusting controller, agent, or proxy status without validating external state and user outcome.
- Sharing credentials, workspaces, caches, or context across tenants and roles without explicit policy.
- Allowing AI-generated plans, policies, or changes to bypass deterministic validation and human authority.
- Declaring recovery after process restart without reconciling orphan effects, leases, artifacts, and evidence.

### Validation and evidence

| Method | Expected evidence |
| --- | --- |
| Examine | Requirements, APIs, identities, state machines, policies, versions, resources, and dependencies for kubernetes-native and cel policy. |
| Observe | Queue, task, controller, model, tool, route, proxy, event, resource, trace, artifact, and user-visible state. |
| Test | Expected, prohibited, malformed, duplicate, overloaded, partitioned, failed, recovery, upgrade, and rollback conditions. |
| Measure | Task success, reliability, latency, throughput, fairness, cost, token and compute use, blast radius, and operator burden. |
| Reconcile | Desired and planned state against actual cloud, cluster, agent, network, data, external-effect, and evidence state. |

## 6.4 Policy lifecycle and conflict

Policy lifecycle and conflict must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Manage ownership, precedence, versions, tests, rollout, audit, enforcement, exception, and retirement. The design should identify authoritative state, workload identities, typed contracts, resource and token budgets, trust boundaries, failure domains, telemetry, rollback, and acceptance evidence.

This guide traces the subject through the developer intent, catalog, template or API, policy, Git source, reconciliation, platform resource, evidence, and user outcome chain. A running agent, reconciled object, successful API call, healthy proxy, or completed function is not proof of the intended outcome. Desired state must be compared with applied state, external effects, negative tests, degraded behavior, causal evidence, cost, and user or mission results.

Production engineering begins with a minimal known-good control plane and explicit invariants. Add one worker class, model route, controller, policy, event source, gateway, or platform feature at a time; preserve before-state evidence; test authorized and prohibited behavior; inject realistic failures; and retain a tested compensation or rollback path.

Troubleshooting locates the first divergence from the expected state machine. Account for stale watches, leases, retries, eventual consistency, duplicated work, cache state, identity rotation, partial external effects, version skew, prompt injection, overload, network partitions, and missing telemetry. Closure requires causal explanation, reproducibility, validated recovery, and residual limitations.

### Engineering practices

- Define authority, ownership, desired state, dependencies, limits, and evidence for policy lifecycle and conflict.
- Use typed APIs, stable identities, explicit state machines, idempotency, and bounded effects.
- Separate control plane, execution plane, data plane, evidence plane, and human decision authority.

Test nominal, prohibited, overloaded, partitioned, partially failed, recovery, upgrade, and rollback cases.

Preserve configuration, policy, model, source, trace, event, artifact, approval, and user-outcome evidence.

Scale through quotas, queues, backpressure, fairness, isolation, canaries, and deterministic fan-in.

## Failure patterns

Treating policy lifecycle and conflict as a feature rather than an end-to-end stateful system.

Scaling worker count or deployment scope before bounding queues, budgets, retries, and fan-in.

Trusting controller, agent, or proxy status without validating external state and user outcome.

Sharing credentials, workspaces, caches, or context across tenants and roles without explicit policy.

Allowing AI-generated plans, policies, or changes to bypass deterministic validation and human authority.

Declaring recovery after process restart without reconciling orphan effects, leases, artifacts, and evidence.

## Validation and evidence

| Method | Expected evidence |
| --- | --- |
| Examine | Requirements, APIs, identities, state machines, policies, versions, resources, and dependencies for policy lifecycle and conflict. |
| Observe | Queue, task, controller, model, tool, route, proxy, event, resource, trace, artifact, and user-visible state. |
| Test | Expected, prohibited, malformed, duplicate, overloaded, partitioned, failed, recovery, upgrade, and rollback conditions. |
| Measure | Task success, reliability, latency, throughput, fairness, cost, token and compute use, blast radius, and operator burden. |
| Reconcile | Desired and planned state against actual cloud, cluster, agent, network, data, external-effect, and evidence state. |

# Chapter 06 laboratory and review

Practical laboratory
Create equivalent policy tests for an infrastructure plan, Kubernetes resource, and platform API request.

## Laboratory evidence package

- Architecture or topology showing boundaries, identities, dependencies, and authoritative inputs.
- Versioned configuration or code with explanatory comments and reproducible setup instructions.
- Nominal-path validation plus at least two injected failure or boundary conditions.
- Structured logs, traces, metrics, packet captures, test output, or other appropriate evidence.
- A concise decision record covering findings, limitations, residual risks, and next actions.

## Frontier extension

Explore policy synthesis from standards with human-reviewed executable controls.

Review gate
Advance only when another engineer can reproduce the result, explain the architecture, identify the failure boundary, and distinguish measured evidence from assumptions or projections.

CHAPTER 07

# Platform security, supply chain, and evidence

Protect the platform because its automation has broad organizational authority.

Chapter operating position
Protect the platform because its automation has broad organizational authority.

## 7.1 Identity and privilege

Identity and privilege must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Use workload identity, JIT administration, scoped controllers, separation, break glass, and audit. The design should identify authoritative state, workload identities, typed contracts, resource and token budgets, trust boundaries, failure domains, telemetry, rollback, and acceptance evidence.

This guide traces the subject through the developer intent, catalog, template or API, policy, Git source, reconciliation, platform resource, evidence, and user outcome chain. A running agent, reconciled object, successful API call, healthy proxy, or completed function is not proof of the intended outcome. Desired state must be compared with applied state, external effects, negative tests, degraded behavior, causal evidence, cost, and user or mission results.

Production engineering begins with a minimal known-good control plane and explicit invariants. Add one worker class, model route, controller, policy, event source, gateway, or platform feature at a time; preserve before-state evidence; test authorized and prohibited behavior; inject realistic failures; and retain a tested compensation or rollback path.

Troubleshooting locates the first divergence from the expected state machine. Account for stale watches, leases, retries, eventual consistency, duplicated work, cache state, identity rotation, partial external effects, version skew, prompt injection, overload, network partitions, and missing telemetry. Closure requires causal explanation, reproducibility, validated recovery, and residual limitations.

### Engineering practices

- Define authority, ownership, desired state, dependencies, limits, and evidence for identity and privilege.
- Use typed APIs, stable identities, explicit state machines, idempotency, and bounded effects.
- Separate control plane, execution plane, data plane, evidence plane, and human decision authority.
- Test nominal, prohibited, overloaded, partitioned, partially failed, recovery, upgrade, and rollback cases.
- Preserve configuration, policy, model, source, trace, event, artifact, approval, and user-outcome evidence.
- Scale through quotas, queues, backpressure, fairness, isolation, canaries, and deterministic fan-in.

### Failure patterns

- Treating identity and privilege as a feature rather than an end-to-end stateful system.
- Scaling worker count or deployment scope before bounding queues, budgets, retries, and fan-in.
- Trusting controller, agent, or proxy status without validating external state and user outcome.
- Sharing credentials, workspaces, caches, or context across tenants and roles without explicit policy.
- Allowing AI-generated plans, policies, or changes to bypass deterministic validation and human authority.
- Declaring recovery after process restart without reconciling orphan effects, leases, artifacts, and evidence.

### Validation and evidence

| Method | Expected evidence |
| --- | --- |
| Examine | Requirements, APIs, identities, state machines, policies, versions, resources, and dependencies for identity and privilege. |
| Observe | Queue, task, controller, model, tool, route, proxy, event, resource, trace, artifact, and user-visible state. |
| Test | Expected, prohibited, malformed, duplicate, overloaded, partitioned, failed, recovery, upgrade, and rollback conditions. |
| Measure | Task success, reliability, latency, throughput, fairness, cost, token and compute use, blast radius, and operator burden. |
| Reconcile | Desired and planned state against actual cloud, cluster, agent, network, data, external-effect, and evidence state. |

## 7.2 Template, plugin, and provider supply chain

Template, plugin, and provider supply chain must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Verify publishers, source, dependencies, images, signatures, provenance, permissions, and updates. The design should identify authoritative state, workload identities, typed contracts, resource and token budgets, trust boundaries, failure domains, telemetry, rollback, and acceptance evidence.

This guide traces the subject through the developer intent, catalog, template or API, policy, Git source, reconciliation, platform resource, evidence, and user outcome chain. A running agent, reconciled object, successful API call, healthy proxy, or completed function is not proof of the intended outcome. Desired state must be compared with applied state, external effects, negative tests, degraded behavior, causal evidence, cost, and user or mission results.

Production engineering begins with a minimal known-good control plane and explicit invariants. Add one worker class, model route, controller, policy, event source, gateway, or platform feature at a time; preserve before-state evidence; test authorized and prohibited behavior; inject realistic failures; and retain a tested compensation or rollback path.

Troubleshooting locates the first divergence from the expected state machine. Account for stale watches, leases, retries, eventual consistency, duplicated work, cache state, identity rotation, partial external effects, version skew, prompt injection, overload, network partitions, and missing telemetry. Closure requires causal explanation, reproducibility, validated recovery, and residual limitations.

### Engineering practices

- Define authority, ownership, desired state, dependencies, limits, and evidence for template, plugin, and provider supply chain.
- Use typed APIs, stable identities, explicit state machines, idempotency, and bounded effects.
- Separate control plane, execution plane, data plane, evidence plane, and human decision authority.
- Test nominal, prohibited, overloaded, partitioned, partially failed, recovery, upgrade, and rollback cases.
- Preserve configuration, policy, model, source, trace, event, artifact, approval, and user-outcome evidence.
- Scale through quotas, queues, backpressure, fairness, isolation, canaries, and deterministic fan-in.

### Implementation sketch

```
platform_request:
  api: ApplicationEnvironment/v2
  owner: team-payments
  environment: production
  composition:
    - kubernetes-namespace
    - workload-identity
    - database
    - gateway-route
    - telemetry
  policies:
    - signed-artifacts
    - restricted-pods
    - encrypted-data
  gitops:
    repository: environments-prod
    promotion: pull-request
  evidence:
    require: [policy-tests, drift-check, rollback-plan]
```

### Failure patterns

- Treating template, plugin, and provider supply chain as a feature rather than an end-to-end stateful system.
- Scaling worker count or deployment scope before bounding queues, budgets, retries, and fan-in.
- Trusting controller, agent, or proxy status without validating external state and user outcome.
- Sharing credentials, workspaces, caches, or context across tenants and roles without explicit policy.
- Allowing AI-generated plans, policies, or changes to bypass deterministic validation and human authority.
- Declaring recovery after process restart without reconciling orphan effects, leases, artifacts, and evidence.

### Validation and evidence

| Method | Expected evidence |
| --- | --- |
| Examine | Requirements, APIs, identities, state machines, policies, versions, resources, and dependencies for template, plugin, and provider supply chain. |
| Observe | Queue, task, controller, model, tool, route, proxy, event, resource, trace, artifact, and user-visible state. |
| Test | Expected, prohibited, malformed, duplicate, overloaded, partitioned, failed, recovery, upgrade, and rollback conditions. |
| Measure | Task success, reliability, latency, throughput, fairness, cost, token and compute use, blast radius, and operator burden. |
| Reconcile | Desired and planned state against actual cloud, cluster, agent, network, data, external-effect, and evidence state. |

## 7.3 Secrets and data boundaries

Secrets and data boundaries must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Broker credentials, isolate tenants, control logs, protect customer data, and rotate trust. The design should identify authoritative state, workload identities, typed contracts, resource and token budgets, trust boundaries, failure domains, telemetry, rollback, and acceptance evidence.

This guide traces the subject through the developer intent, catalog, template or API, policy, Git source, reconciliation, platform resource, evidence, and user outcome chain. A running agent, reconciled object, successful API call, healthy proxy, or completed function is not proof of the intended outcome. Desired state must be compared with applied state, external effects, negative tests, degraded behavior, causal evidence, cost, and user or mission results.

Production engineering begins with a minimal known-good control plane and explicit invariants. Add one worker class, model route, controller, policy, event source, gateway, or platform feature at a time; preserve before-state evidence; test authorized and prohibited behavior; inject realistic failures; and retain a tested compensation or rollback path.

Troubleshooting locates the first divergence from the expected state machine. Account for stale watches, leases, retries, eventual consistency, duplicated work, cache state, identity rotation, partial external effects, version skew, prompt injection, overload, network partitions, and missing telemetry. Closure requires causal explanation, reproducibility, validated recovery, and residual limitations.

### Engineering practices

- Define authority, ownership, desired state, dependencies, limits, and evidence for secrets and data boundaries.
- Use typed APIs, stable identities, explicit state machines, idempotency, and bounded effects.
- Separate control plane, execution plane, data plane, evidence plane, and human decision authority.
- Test nominal, prohibited, overloaded, partitioned, partially failed, recovery, upgrade, and rollback cases.
- Preserve configuration, policy, model, source, trace, event, artifact, approval, and user-outcome evidence.
- Scale through quotas, queues, backpressure, fairness, isolation, canaries, and deterministic fan-in.

### Failure patterns

- Treating secrets and data boundaries as a feature rather than an end-to-end stateful system.
- Scaling worker count or deployment scope before bounding queues, budgets, retries, and fan-in.
- Trusting controller, agent, or proxy status without validating external state and user outcome.

Sharing credentials, workspaces, caches, or context across tenants and roles without explicit policy.

Allowing AI-generated plans, policies, or changes to bypass deterministic validation and human authority.

Declaring recovery after process restart without reconciling orphan effects, leases, artifacts, and evidence.

### Validation and evidence

| Method | Expected evidence |
| --- | --- |
| Examine | Requirements, APIs, identities, state machines, policies, versions, resources, and dependencies for secrets and data boundaries. |
| Observe | Queue, task, controller, model, tool, route, proxy, event, resource, trace, artifact, and user-visible state. |
| Test | Expected, prohibited, malformed, duplicate, overloaded, partitioned, failed, recovery, upgrade, and rollback conditions. |
| Measure | Task success, reliability, latency, throughput, fairness, cost, token and compute use, blast radius, and operator burden. |
| Reconcile | Desired and planned state against actual cloud, cluster, agent, network, data, external-effect, and evidence state. |

## 7.4 Evidence and control validation

Evidence and control validation must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Record requests, policy decisions, changes, artifacts, reconciliations, tests, and user outcomes. The design should identify authoritative state, workload identities, typed contracts, resource and token budgets, trust boundaries, failure domains, telemetry, rollback, and acceptance evidence.

This guide traces the subject through the developer intent, catalog, template or API, policy, Git source, reconciliation, platform resource, evidence, and user outcome chain. A running agent, reconciled object, successful API call, healthy proxy, or completed function is not proof of the intended outcome. Desired state must be compared with applied state, external effects, negative tests, degraded behavior, causal evidence, cost, and user or mission results.

Production engineering begins with a minimal known-good control plane and explicit invariants. Add one worker class, model route, controller, policy, event source, gateway, or platform feature at a time; preserve before-state evidence; test authorized and prohibited behavior; inject realistic failures; and retain a tested compensation or rollback path.

Troubleshooting locates the first divergence from the expected state machine. Account for stale watches, leases, retries, eventual consistency, duplicated work, cache state, identity rotation, partial external effects, version skew, prompt injection, overload, network partitions, and missing telemetry. Closure requires causal explanation, reproducibility, validated recovery, and residual limitations.

### Engineering practices

Define authority, ownership, desired state, dependencies, limits, and evidence for evidence and control validation.

Use typed APIs, stable identities, explicit state machines, idempotency, and bounded effects.

Separate control plane, execution plane, data plane, evidence plane, and human decision authority.

Test nominal, prohibited, overloaded, partitioned, partially failed, recovery, upgrade, and rollback cases.

Preserve configuration, policy, model, source, trace, event, artifact, approval, and user-outcome evidence.

Scale through quotas, queues, backpressure, fairness, isolation, canaries, and deterministic fan-in.

### Failure patterns

Treating evidence and control validation as a feature rather than an end-to-end stateful system.

Scaling worker count or deployment scope before bounding queues, budgets, retries, and fan-in.

Trusting controller, agent, or proxy status without validating external state and user outcome.

Sharing credentials, workspaces, caches, or context across tenants and roles without explicit policy.

Allowing AI-generated plans, policies, or changes to bypass deterministic validation and human authority.

Declaring recovery after process restart without reconciling orphan effects, leases, artifacts, and evidence.

## Validation and evidence

| Method | Expected evidence |
| --- | --- |
| Examine | Requirements, APIs, identities, state machines, policies, versions, resources, and dependencies for evidence and control validation. |
| Observe | Queue, task, controller, model, tool, route, proxy, event, resource, trace, artifact, and user-visible state. |
| Test | Expected, prohibited, malformed, duplicate, overloaded, partitioned, failed, recovery, upgrade, and rollback conditions. |
| Measure | Task success, reliability, latency, throughput, fairness, cost, token and compute use, blast radius, and operator burden. |
| Reconcile | Desired and planned state against actual cloud, cluster, agent, network, data, external-effect, and evidence state. |

# Chapter 07 laboratory and review

Practical laboratory
Threat-model the portal, scaffolder, GitOps controller, Crossplane providers, policy engine, and cloud credentials.

## Laboratory evidence package

- Architecture or topology showing boundaries, identities, dependencies, and authoritative inputs.
- Versioned configuration or code with explanatory comments and reproducible setup instructions.
- Nominal-path validation plus at least two injected failure or boundary conditions.
- Structured logs, traces, metrics, packet captures, test output, or other appropriate evidence.
- A concise decision record covering findings, limitations, residual risks, and next actions.

## Frontier extension

Explore confidential platform control planes and signed automation receipts.

Review gate
Advance only when another engineer can reproduce the result, explain the architecture, identify the failure boundary, and distinguish measured evidence from assumptions or projections.

CHAPTER 08

# Operations, adoption, and continuous evolution

Operate the platform as a reliable product and avoid becoming a new bottleneck.

Chapter operating position
Operate the platform as a reliable product and avoid becoming a new bottleneck.

## 8.1 SLOs and support

SLOs and support must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Define platform API, provisioning, deployment, catalog, pipeline, policy, and incident objectives. The design should identify authoritative state, workload identities, typed contracts, resource and token budgets, trust boundaries, failure domains, telemetry, rollback, and acceptance evidence.

This guide traces the subject through the developer intent, catalog, template or API, policy, Git source, reconciliation, platform resource, evidence, and user outcome chain. A running agent, reconciled object, successful API call, healthy proxy, or completed function is not proof of the intended outcome. Desired state must be compared with applied state, external effects, negative tests, degraded behavior, causal evidence, cost, and user or mission results.

Production engineering begins with a minimal known-good control plane and explicit invariants. Add one worker class, model route, controller, policy, event source, gateway, or platform feature at a time; preserve before-state evidence; test authorized and prohibited behavior; inject realistic failures; and retain a tested compensation or rollback path.

Troubleshooting locates the first divergence from the expected state machine. Account for stale watches, leases, retries, eventual consistency, duplicated work, cache state, identity rotation, partial external effects, version skew, prompt injection, overload, network partitions, and missing telemetry. Closure requires causal explanation, reproducibility, validated recovery, and residual limitations.

### Engineering practices

- Define authority, ownership, desired state, dependencies, limits, and evidence for slos and support.
- Use typed APIs, stable identities, explicit state machines, idempotency, and bounded effects.
- Separate control plane, execution plane, data plane, evidence plane, and human decision authority.
- Test nominal, prohibited, overloaded, partitioned, partially failed, recovery, upgrade, and rollback cases.
- Preserve configuration, policy, model, source, trace, event, artifact, approval, and user-outcome evidence.
- Scale through quotas, queues, backpressure, fairness, isolation, canaries, and deterministic fan-in.

### Failure patterns

- Treating slos and support as a feature rather than an end-to-end stateful system.
- Scaling worker count or deployment scope before bounding queues, budgets, retries, and fan-in.
- Trusting controller, agent, or proxy status without validating external state and user outcome.
- Sharing credentials, workspaces, caches, or context across tenants and roles without explicit policy.
- Allowing AI-generated plans, policies, or changes to bypass deterministic validation and human authority.
- Declaring recovery after process restart without reconciling orphan effects, leases, artifacts, and evidence.

### Validation and evidence

| Method | Expected evidence |
| --- | --- |
| Examine | Requirements, APIs, identities, state machines, policies, versions, resources, and dependencies for slos and support. |
| Observe | Queue, task, controller, model, tool, route, proxy, event, resource, trace, artifact, and user-visible state. |
| Test | Expected, prohibited, malformed, duplicate, overloaded, partitioned, failed, recovery, upgrade, and rollback conditions. |
| Measure | Task success, reliability, latency, throughput, fairness, cost, token and compute use, blast radius, and operator burden. |
| Reconcile | Desired and planned state against actual cloud, cluster, agent, network, data, external-effect, and evidence state. |

## 8.2 Feedback and discovery

Feedback and discovery must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Use analytics, interviews, support cases, failed journeys, product research, and community. The design should identify authoritative state, workload identities, typed contracts, resource and token budgets, trust boundaries, failure domains, telemetry, rollback, and acceptance evidence.

This guide traces the subject through the developer intent, catalog, template or API, policy, Git source, reconciliation, platform resource, evidence, and user outcome chain. A running agent, reconciled object, successful API call, healthy proxy, or completed function is not proof of the intended outcome. Desired state must be compared with applied state, external effects, negative tests, degraded behavior, causal evidence, cost, and user or mission results.

Production engineering begins with a minimal known-good control plane and explicit invariants. Add one worker class, model route, controller, policy, event source, gateway, or platform feature at a time; preserve before-state evidence; test authorized and prohibited behavior; inject realistic failures; and retain a tested compensation or rollback path.

Troubleshooting locates the first divergence from the expected state machine. Account for stale watches, leases, retries, eventual consistency, duplicated work, cache state, identity rotation, partial external effects, version skew, prompt injection, overload, network partitions, and missing telemetry. Closure requires causal explanation, reproducibility, validated recovery, and residual limitations.

### Engineering practices

- Define authority, ownership, desired state, dependencies, limits, and evidence for feedback and discovery.
- Use typed APIs, stable identities, explicit state machines, idempotency, and bounded effects.
- Separate control plane, execution plane, data plane, evidence plane, and human decision authority.
- Test nominal, prohibited, overloaded, partitioned, partially failed, recovery, upgrade, and rollback cases.
- Preserve configuration, policy, model, source, trace, event, artifact, approval, and user-outcome evidence.
- Scale through quotas, queues, backpressure, fairness, isolation, canaries, and deterministic fan-in.

### Implementation sketch

```
platform_request:
  api: ApplicationEnvironment/v2
  owner: team-payments
  environment: production
  composition:
    - kubernetes-namespace
    - workload-identity
    - database
    - gateway-route
    - telemetry
  policies:
    - signed-artifacts
    - restricted-pods
    - encrypted-data
  gitops:
    repository: environments-prod
    promotion: pull-request
  evidence:
    require: [policy-tests, drift-check, rollback-plan]
```

### Failure patterns

- Treating feedback and discovery as a feature rather than an end-to-end stateful system.
- Scaling worker count or deployment scope before bounding queues, budgets, retries, and fan-in.
- Trusting controller, agent, or proxy status without validating external state and user outcome.
- Sharing credentials, workspaces, caches, or context across tenants and roles without explicit policy.
- Allowing AI-generated plans, policies, or changes to bypass deterministic validation and human authority.
- Declaring recovery after process restart without reconciling orphan effects, leases, artifacts, and evidence.

### Validation and evidence

| Method | Expected evidence |
| --- | --- |
| Examine | Requirements, APIs, identities, state machines, policies, versions, resources, and dependencies for feedback and discovery. |
| Observe | Queue, task, controller, model, tool, route, proxy, event, resource, trace, artifact, and user-visible state. |
| Test | Expected, prohibited, malformed, duplicate, overloaded, partitioned, failed, recovery, upgrade, and rollback conditions. |
| Measure | Task success, reliability, latency, throughput, fairness, cost, token and compute use, blast radius, and operator burden. |
| Reconcile | Desired and planned state against actual cloud, cluster, agent, network, data, external-effect, and evidence state. |

## 8.3 Migration and enablement

Migration and enablement must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Onboard teams, import existing systems, document changes, train users, and retire old paths. The design should identify authoritative state, workload identities, typed contracts, resource and token budgets, trust boundaries, failure domains, telemetry, rollback, and acceptance evidence.

This guide traces the subject through the developer intent, catalog, template or API, policy, Git source, reconciliation, platform resource, evidence, and user outcome chain. A running agent, reconciled object, successful API call, healthy proxy, or completed function is not proof of the intended outcome. Desired state must be compared with applied state, external effects, negative tests, degraded behavior, causal evidence, cost, and user or mission results.

Production engineering begins with a minimal known-good control plane and explicit invariants. Add one worker class, model route, controller, policy, event source, gateway, or platform feature at a time; preserve before-state evidence; test authorized and prohibited behavior; inject realistic failures; and retain a tested compensation or rollback path.

Troubleshooting locates the first divergence from the expected state machine. Account for stale watches, leases, retries, eventual consistency, duplicated work, cache state, identity rotation, partial external effects, version skew, prompt injection, overload, network partitions, and missing telemetry. Closure requires causal explanation, reproducibility, validated recovery, and residual limitations.

### Engineering practices

- Define authority, ownership, desired state, dependencies, limits, and evidence for migration and enablement.
- Use typed APIs, stable identities, explicit state machines, idempotency, and bounded effects.
- Separate control plane, execution plane, data plane, evidence plane, and human decision authority.
- Test nominal, prohibited, overloaded, partitioned, partially failed, recovery, upgrade, and rollback cases.
- Preserve configuration, policy, model, source, trace, event, artifact, approval, and user-outcome evidence.
- Scale through quotas, queues, backpressure, fairness, isolation, canaries, and deterministic fan-in.

### Failure patterns

- Treating migration and enablement as a feature rather than an end-to-end stateful system.
- Scaling worker count or deployment scope before bounding queues, budgets, retries, and fan-in.
- Trusting controller, agent, or proxy status without validating external state and user outcome.

Sharing credentials, workspaces, caches, or context across tenants and roles without explicit policy.

Allowing AI-generated plans, policies, or changes to bypass deterministic validation and human authority.

Declaring recovery after process restart without reconciling orphan effects, leases, artifacts, and evidence.

### Validation and evidence

| Method | Expected evidence |
|---|---|
| Examine | Requirements, APIs, identities, state machines, policies, versions, resources, and dependencies for migration and enablement. |
| Observe | Queue, task, controller, model, tool, route, proxy, event, resource, trace, artifact, and user-visible state. |
| Test | Expected, prohibited, malformed, duplicate, overloaded, partitioned, failed, recovery, upgrade, and rollback conditions. |
| Measure | Task success, reliability, latency, throughput, fairness, cost, token and compute use, blast radius, and operator burden. |
| Reconcile | Desired and planned state against actual cloud, cluster, agent, network, data, external-effect, and evidence state. |

## 8.4 Platform roadmap and economics

Platform roadmap and economics must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Prioritize capabilities by user value, risk, leverage, cost, capacity, and organizational readiness. The design should identify authoritative state, workload identities, typed contracts, resource and token budgets, trust boundaries, failure domains, telemetry, rollback, and acceptance evidence.

This guide traces the subject through the developer intent, catalog, template or API, policy, Git source, reconciliation, platform resource, evidence, and user outcome chain. A running agent, reconciled object, successful API call, healthy proxy, or completed function is not proof of the intended outcome. Desired state must be compared with applied state, external effects, negative tests, degraded behavior, causal evidence, cost, and user or mission results.

Production engineering begins with a minimal known-good control plane and explicit invariants. Add one worker class, model route, controller, policy, event source, gateway, or platform feature at a time; preserve before-state evidence; test authorized and prohibited behavior; inject realistic failures; and retain a tested compensation or rollback path.

Troubleshooting locates the first divergence from the expected state machine. Account for stale watches, leases, retries, eventual consistency, duplicated work, cache state, identity rotation, partial external effects, version skew, prompt injection, overload, network partitions, and missing telemetry. Closure requires causal explanation, reproducibility, validated recovery, and residual limitations.

### Engineering practices

Define authority, ownership, desired state, dependencies, limits, and evidence for platform roadmap and economics.

Use typed APIs, stable identities, explicit state machines, idempotency, and bounded effects.

Separate control plane, execution plane, data plane, evidence plane, and human decision authority.

Test nominal, prohibited, overloaded, partitioned, partially failed, recovery, upgrade, and rollback cases.

Preserve configuration, policy, model, source, trace, event, artifact, approval, and user-outcome evidence.

Scale through quotas, queues, backpressure, fairness, isolation, canaries, and deterministic fan-in.

### Failure patterns

Treating platform roadmap and economics as a feature rather than an end-to-end stateful system.

Scaling worker count or deployment scope before bounding queues, budgets, retries, and fan-in.

Trusting controller, agent, or proxy status without validating external state and user outcome.

Sharing credentials, workspaces, caches, or context across tenants and roles without explicit policy.

Allowing AI-generated plans, policies, or changes to bypass deterministic validation and human authority.

Declaring recovery after process restart without reconciling orphan effects, leases, artifacts, and evidence.

## Validation and evidence

| Method | Expected evidence |
| --- | --- |
| Examine | Requirements, APIs, identities, state machines, policies, versions, resources, and dependencies for platform roadmap and economics. |
| Observe | Queue, task, controller, model, tool, route, proxy, event, resource, trace, artifact, and user-visible state. |
| Test | Expected, prohibited, malformed, duplicate, overloaded, partitioned, failed, recovery, upgrade, and rollback conditions. |
| Measure | Task success, reliability, latency, throughput, fairness, cost, token and compute use, blast radius, and operator burden. |
| Reconcile | Desired and planned state against actual cloud, cluster, agent, network, data, external-effect, and evidence state. |

# Chapter 08 laboratory and review

Practical laboratory
Create a twelve-month platform roadmap from developer evidence and define which capabilities should not be centralized.

## Laboratory evidence package

- Architecture or topology showing boundaries, identities, dependencies, and authoritative inputs.
- Versioned configuration or code with explanatory comments and reproducible setup instructions.
- Nominal-path validation plus at least two injected failure or boundary conditions.
- Structured logs, traces, metrics, packet captures, test output, or other appropriate evidence.
- A concise decision record covering findings, limitations, residual risks, and next actions.

## Frontier extension

Explore agentic platform operations with human product governance and automatically verified changes.

Review gate
Advance only when another engineer can reproduce the result, explain the architecture, identify the failure boundary, and distinguish measured evidence from assumptions or projections.

# Integrated learning roadmap

The guide is intended to be revisited. First establish fluency, then build a small implementation, operate it under failure, automate repeatable work, and finally evaluate emerging techniques against a stable baseline. Progress is demonstrated through evidence rather than reading completion alone.

| Stage | Demonstrated capability |
| --- | --- |
| Foundation | Explain the system from first principles and trace its critical path without relying on product terminology. |
| Build | Implement the smallest representative system with explicit contracts, identities, telemetry, and tests. |
| Operate | Diagnose injected failures, recover safely, and preserve evidence of the causal chain. |
| Automate | Convert a proven manual workflow into bounded, idempotent, observable automation. |
| Architect | Design for scale, resilience, security, interoperability, and lifecycle change. |
| Explore | Evaluate frontier techniques using stated hypotheses, limitations, and adoption gates. |

# Source authority register

Source policy

Official standards, specifications, and project documentation remain with their issuing organizations. Mythos Systems provides original engineering interpretation, synthesis, implementation guidance, and relationship mapping. Source verification dates do not guarantee that an authority has not changed since review.

01. CNCF - Platform Engineering Technical Community Group
Role: Cloud-native platform-engineering practices and reference material.
Verified: 2026-07-26
Official source: https://tag-app-delivery.cncf.io/wgs/platforms/

02. Backstage - Backstage Software Catalog
Role: Software ownership and metadata catalog based on source-controlled entity definitions.
Verified: 2026-07-26
Official source: https://backstage.io/docs/features/software-catalog/

03. Backstage - Software Templates and Scaffolder
Role: Self-service software templates, actions, workflows, and authorization.
Verified: 2026-07-26
Official source: https://backstage.io/docs/features/software-templates/

04. Crossplane - Crossplane Documentation
Role: Control-plane framework for custom APIs, composition, managed resources, and platform engineering.
Verified: 2026-07-26
Official source: https://docs.crossplane.io/latest/

05. Crossplane - What's New in Crossplane v2
Role: Current v2 architecture, namespaced resources, composition, and operational workflows.
Verified: 2026-07-26
Official source: https://docs.crossplane.io/latest/whats-new/

06. Argo Project - Argo CD Documentation
Role: Declarative GitOps continuous delivery and reconciliation.
Verified: 2026-07-26
Official source: https://argo-cd.readthedocs.io/en/stable/

07. Flux - Flux Documentation
Role: GitOps toolkit, source reconciliation, delivery, notifications, and controllers.
Verified: 2026-07-26
Official source: https://fluxcd.io/flux/

08. Open Policy Agent - Open Policy Agent Documentation
Role: General-purpose policy engine and Rego policy language.
Verified: 2026-07-26
Official source: https://www.openpolicyagent.org/docs

09. Kyverno - Kyverno Documentation
Role: Kubernetes-native and CEL-oriented policy-as-code lifecycle.
Verified: 2026-07-26
Official source: https://kyverno.io/docs/

## Authority application and refresh triggers

| Authority class | Required library action |
| --- | --- |
| Protocols and specifications | Recheck interoperability, security considerations, defaults, and migration guidance when a referenced specification changes. |
| Security and assurance frameworks | Update control mappings, evidence expectations, risk language, and assessment procedures after material framework revision. |
| Languages, runtimes, and projects | Re-run examples, tests, deprecation checks, and operational recommendations against supported releases. |
| Benchmarks and evaluations | Re-execute claims when workloads, methods, models, compilers, drivers, hardware, or scoring rules change. |
| Operational evidence | Incorporate validated incidents, failure tests, reader corrections, and measured implementation outcomes into the next revision. |

# Limitations, freshness, and revision control

| Boundary | Statement |
| --- | --- |
| Publication limitation | This guide synthesizes broad engineering practice. It does not replace product documentation, legal advice, safety certification, or environment-specific design review. |
| Evidence limitation | Not every pattern is benchmarked in every environment. Examples illustrate engineering methods and must be validated against actual versions, workloads, hardware, policy, and failure conditions. |
| Freshness limitation | Vendor behavior, software libraries, AI models, security threats, and emerging standards can change quickly. Volatile claims require source revalidation before operational adoption. |
| Adoption limitation | Frontier content is not an implicit adoption recommendation. Adoption requires defined value, qualification gates, controlled proof, rollback, ownership, and residual-risk acceptance. |
| Status boundary | Candidate Focused Field Guide describes the publication, not certification of a product, implementation, engineer, or organization. |

## Revision record

| Version | Revision statement |
| --- | --- |
| 1.0.0-candidate | Initial candidate living-guide edition. Published 2026-07-26; scheduled review 2026-10-26. |

## Search and relationship metadata

| Dimension | Engineering position |
| --- | --- |
| Primary domain | MYTHOS-CLD - Cloud, Platform & Distributed Systems |
| Secondary domain | MYTHOS-SWE; MYTHOS-SEC; MYTHOS-DAT |
| Publication class | Field Guide / Canonical Living Overview Guide |
| Skill levels | L1 Foundational; L2 Practitioner; L3 Advanced; L4 Frontier |
| Tags | platform engineering, internal developer platform, GitOps, policy as code, Backstage, Crossplane, Argo CD, Flux, OPA, Kyverno |
| Canonical route | /library/field-guides/mythos-fg-cld-0003/ |

Living publication

Corrections, expanded laboratories, improved diagrams, authority changes, and new engineering evidence should revise this permanent publication rather than create disconnected replacements.