



CANONICAL LIVING OVERVIEW GUIDE

Agent Harnesses, Multi-Harness Fabrics, and Runtime Isolation

A frontier runtime guide defining agent harnesses as governed execution authorities and multi-harness fabrics as distributed systems for routing missions across isolated workspaces, tools, models, credentials, memory, sandboxes, and evidence planes.

PERMANENT PUBLICATION IDENTITY

Agent Harnesses, Multi-Harness Fabrics, and Runtime Isolation

Document ID
MYTHOS-FG-AI-0010

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

Reader orientation

Dimension	Engineering position
Primary domain	MYTHOS-AI - Artificial Intelligence & Agentic Systems
Secondary domain	MYTHOS-SEC; MYTHOS-SWE; MYTHOS-CLD
Audience	Agent-runtime architects, AI platform engineers, security engineers, software-development platform teams, and advanced automation practitioners.
Prerequisites	Agentic systems, operating systems, containers, distributed systems, identity, software delivery, and security architecture.
Publisher / owner	Mythos Systems / Aaron Stovall
Public classification	Public technical guidance

Expected outcomes

- Define a harness contract spanning model, context, tools, workspace, policy, state, telemetry, and lifecycle.
- Compose multiple heterogeneous harnesses into a governed fabric with typed handoffs and deterministic mission state.
- Isolate files, networks, credentials, processes, accelerators, models, and tenants according to risk.
- Recover from harness, tool, model, worker, and orchestration failure without losing evidence or authority.
- Evaluate fabric safety, throughput, reproducibility, blast radius, cost, and verified completion.

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

How to use this guide	3
Contents	3
Chapter 01 - Harness anatomy and execution contract	5
Chapter 01 laboratory and review	10
Chapter 02 - Multi-harness fabric architecture	11
Chapter 02 laboratory and review	16
Chapter 03 - Isolation and sandbox hierarchy	17
Chapter 03 laboratory and review	22
Chapter 04 - Identity, credentials, and capability authority	23
Chapter 04 laboratory and review	28
Chapter 05 - Mission state, checkpoints, and event sourcing	29
Chapter 05 laboratory and review	34

Chapter 06 - Scheduling, capacity, and placement	35
Chapter 06 laboratory and review	40
Chapter 07 - Failure containment, recovery, and compensation	41
Chapter 07 laboratory and review	46
Chapter 08 - Fabric assurance, observability, and evolution	47
Chapter 08 laboratory and review	52
Integrated learning roadmap	53
Source authority register	54
Limitations, freshness, and revision control	56

CHAPTER 01

Harness anatomy and execution contract

Define the boundary that converts a model into an operational worker.

Chapter operating position

Define the boundary that converts a model into an operational worker.



1.1 Model and provider adapter

Model and provider adapter must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Normalize messages, tool schemas, streaming, structured output, token accounting, errors, and provider limits. 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 mission contract, fabric control plane, harness placement, context, capability grant, execution, checkpoint, evidence, and verified 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 model and provider adapter.

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 model and provider adapter 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 model and provider adapter.
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 Context and memory adapter

Context and memory adapter must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Provide task state, source-grounded context, private scratch, shared memory, compaction, and freshness. 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 mission contract, fabric control plane, harness placement, context, capability grant, execution, checkpoint, evidence, and verified 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 context and memory adapter.
- 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

```
mission:
  id: mission-042
  state: admitted
  harnesses:
    planner:
      model_class: frontier-reasoner
      capabilities: [read:repo, write:plan]
    implementer:
      isolation: microvm
      capabilities: [read:repo, write:worktree, execute:test]
    verifier:
      isolation: separate-image
      capabilities: [read:candidate, execute:hidden-tests]
  fan_in:
    requires: [signed-plan, candidate-patch, test-evidence, independent-review]
  production_effects:
    default: deny
```

Failure patterns

- Treating context and memory adapter 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 context and memory adapter.
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 Tool and capability adapter

Tool and capability adapter must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Expose least-privilege operations with typed arguments, side effects, approvals, cancellation, and evidence. 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 mission contract, fabric control plane, harness placement, context, capability grant, execution, checkpoint, evidence, and verified 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 tool and capability adapter.
- 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 tool and capability adapter 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 tool and capability adapter.
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 Workspace and lifecycle

Workspace and lifecycle must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Create isolated files, repository state, processes, checkpoints, cleanup, retention, and terminal 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 mission contract, fabric control plane, harness placement, context, capability grant, execution, checkpoint, evidence, and verified 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 workspace and lifecycle.

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 workspace and lifecycle 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 workspace and lifecycle.
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

Implement a minimal harness contract and prove that an equivalent mission can run through two model adapters without changing tool authority.

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 formally specified harness interfaces and proof-carrying execution.

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

Multi-harness fabric architecture

Coordinate heterogeneous harnesses without collapsing them into one privileged monolith.

Chapter operating position

Coordinate heterogeneous harnesses without collapsing them into one privileged monolith.

2.1 Fabric control plane

Fabric control plane must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Manage mission graphs, scheduling, identities, policies, budgets, capability catalogs, state, and approvals. 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 mission contract, fabric control plane, harness placement, context, capability grant, execution, checkpoint, evidence, and verified 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 fabric control plane.

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 fabric control plane 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 fabric control plane.
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 Harness data plane

Harness data plane must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Execute bounded work, emit observations, stage artifacts, report health, and accept cancellation or compensation. 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 mission contract, fabric control plane, harness placement, context, capability grant, execution, checkpoint, evidence, and verified 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 harness data plane.
- 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

```
mission:
  id: mission-042
  state: admitted
  harnesses:
    planner:
      model_class: frontier-reasoner
      capabilities: [read:repo, write:plan]
    implementer:
      isolation: microvm
      capabilities: [read:repo, write:worktree, execute:test]
    verifier:
      isolation: separate-image
      capabilities: [read:candidate, execute:hidden-tests]
  fan_in:
    requires: [signed-plan, candidate-patch, test-evidence, independent-review]
  production_effects:
    default: deny
```

Failure patterns

Treating harness data plane 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 harness data plane.
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 Typed handoffs and artifacts

Typed handoffs and artifacts must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Exchange task contracts, claims, evidence, patches, tests, plans, models, and unresolved conditions. 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 mission contract, fabric control plane, harness placement, context, capability grant, execution, checkpoint, evidence, and verified 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 typed handoffs and artifacts.

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 typed handoffs and artifacts 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 typed handoffs and artifacts.
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 Federation and interoperability

Federation and interoperability must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Bridge local harnesses, remote agents, MCP capabilities, A2A peers, and human operators. 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 mission contract, fabric control plane, harness placement, context, capability grant, execution, checkpoint, evidence, and verified 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 federation and interoperability.

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 federation and interoperability 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 federation and interoperability.
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

Build a three-harness fabric with research, implementation, and verification roles and a deterministic fan-in contract.

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 federated multi-organization fabrics with A2A discovery and signed capability advertisements.

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

Isolation and sandbox hierarchy

Select isolation according to code, data, tool, model, and infrastructure risk.

Chapter operating position

Select isolation according to code, data, tool, model, and infrastructure risk.



3.1 Process and user isolation

Process and user isolation must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Use separate identities, environment, resource limits, syscall policy, file roots, and process trees. 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 mission contract, fabric control plane, harness placement, context, capability grant, execution, checkpoint, evidence, and verified 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 process and user isolation.

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 process and user isolation 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 process and user isolation.
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 Container, microVM, and WASM isolation

Container, microVM, and WASM isolation must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Compare startup, kernel sharing, device access, networking, snapshotting, and escape risk. 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 mission contract, fabric control plane, harness placement, context, capability grant, execution, checkpoint, evidence, and verified 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 container, microvm, and wasm isolation.
- 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

```
mission:
  id: mission-042
  state: admitted
  harnesses:
    planner:
      model_class: frontier-reasoner
      capabilities: [read:repo, write:plan]
    implementer:
      isolation: microvm
      capabilities: [read:repo, write:worktree, execute:test]
    verifier:
      isolation: separate-image
      capabilities: [read:candidate, execute:hidden-tests]
  fan_in:
    requires: [signed-plan, candidate-patch, test-evidence, independent-review]
  production_effects:
    default: deny
```

Failure patterns

Treating container, microvm, and wasm isolation 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 container, microvm, and wasm isolation.
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 Workspace and repository isolation

Workspace and repository isolation must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Use branches, worktrees, overlays, content-addressed snapshots, locks, and merge boundaries. 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 mission contract, fabric control plane, harness placement, context, capability grant, execution, checkpoint, evidence, and verified 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 workspace and repository isolation.

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 workspace and repository isolation 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 workspace and repository isolation.
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 Accelerator and model isolation

Accelerator and model isolation must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Control GPU partitions, memory, model weights, adapters, caches, prompts, and cross-tenant reuse. 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 mission contract, fabric control plane, harness placement, context, capability grant, execution, checkpoint, evidence, and verified 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 accelerator and model isolation.

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 accelerator and model isolation 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 accelerator and model isolation.
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

Run the same untrusted coding task under process, container, and microVM threat models and document residual exposure.

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 harnesses with attested code, encrypted memory, and policy-bound key release.

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

Identity, credentials, and capability authority

Bind every action to a workload identity and explicit grant.

Chapter operating position

Bind every action to a workload identity and explicit grant.



4.1 Harness and worker identity

Harness and worker identity must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Issue short-lived workload identities tied to runtime, mission, tenant, role, and environment. 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 mission contract, fabric control plane, harness placement, context, capability grant, execution, checkpoint, evidence, and verified 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 harness and worker identity.

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 harness and worker identity 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 harness and worker identity.
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 Credential brokering

Credential brokering must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Mint scoped tokens just in time, inject them without model visibility, rotate, revoke, 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 mission contract, fabric control plane, harness placement, context, capability grant, execution, checkpoint, evidence, and verified 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 credential brokering.
- 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

```
mission:
  id: mission-042
  state: admitted
  harnesses:
    planner:
      model_class: frontier-reasoner
      capabilities: [read:repo, write:plan]
    implementer:
      isolation: microvm
      capabilities: [read:repo, write:worktree, execute:test]
    verifier:
      isolation: separate-image
      capabilities: [read:candidate, execute:hidden-tests]
  fan_in:
    requires: [signed-plan, candidate-patch, test-evidence, independent-review]
  production_effects:
    default: deny
```

Failure patterns

- Treating credential brokering 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 credential brokering.
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 Capability policy

Capability policy must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Limit tool, operation, target, data, network, duration, rate, cost, and side-effect class. 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 mission contract, fabric control plane, harness placement, context, capability grant, execution, checkpoint, evidence, and verified 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 capability 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 capability 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 capability 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.

4.4 Delegation and human approval

Delegation and human approval must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Constrain subdelegation, require step-up approval, record intent, and prevent confused-deputy behavior. 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 mission contract, fabric control plane, harness placement, context, capability grant, execution, checkpoint, evidence, and verified 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 delegation and human approval.

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 delegation and human approval 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 delegation and human approval.
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 credential broker that permits read-only analysis and canary deployment but denies secret retrieval and production mutation.

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 cryptographic capabilities bound to signed mission plans and attested harness measurements.

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

Mission state, checkpoints, and event sourcing

Preserve authoritative progress independently of any one model context.

Chapter operating position

Preserve authoritative progress independently of any one model context.



5.1 Mission and task state machine

Mission and task state machine must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Represent queued, admitted, running, waiting, reviewing, approved, compensating, complete, and failed states. 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 mission contract, fabric control plane, harness placement, context, capability grant, execution, checkpoint, evidence, and verified 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 mission and task state machine.

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 mission and task state machine 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 mission and task state machine.
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 Event-sourced trajectory

Event-sourced trajectory must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Record observation, plan, action, tool result, artifact, approval, state transition, cost, and causal 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 mission contract, fabric control plane, harness placement, context, capability grant, execution, checkpoint, evidence, and verified 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 event-sourced trajectory.

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

```
mission:
  id: mission-042
  state: admitted
  harnesses:
    planner:
      model_class: frontier-reasoner
      capabilities: [read:repo, write:plan]
    implementer:
      isolation: microvm
      capabilities: [read:repo, write:worktree, execute:test]
    verifier:
      isolation: separate-image
      capabilities: [read:candidate, execute:hidden-tests]
  fan_in:
    requires: [signed-plan, candidate-patch, test-evidence, independent-review]
  production_effects:
    default: deny
```

Failure patterns

- Treating event-sourced trajectory 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 event-sourced trajectory.
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 Checkpoint and resume

Checkpoint and resume must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Persist repository, environment, memory, pending work, leases, and evidence for another harness to resume. 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 mission contract, fabric control plane, harness placement, context, capability grant, execution, checkpoint, evidence, and verified 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 checkpoint and resume.
- 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 checkpoint and resume 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 checkpoint and resume.
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 Consistency and conflict

Consistency and conflict must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Use optimistic concurrency, leases, compare-and-swap, deduplication, merge policy, and human arbitration. 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 mission contract, fabric control plane, harness placement, context, capability grant, execution, checkpoint, evidence, and verified 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 consistency 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 consistency 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 consistency 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 05 laboratory and review

Practical laboratory

Interrupt a multi-step mission during tool execution and resume it on another harness without repeating an external side effect.

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 agent missions and verifiable replay across heterogeneous runtimes.

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

Scheduling, capacity, and placement

Place work according to capability, locality, risk, resources, and deadlines.

Chapter operating position

Place work according to capability, locality, risk, resources, and deadlines.

6.1 Capability-aware scheduling

Capability-aware scheduling must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Match task requirements to model, tools, operating system, accelerator, data location, and permissions. 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 mission contract, fabric control plane, harness placement, context, capability grant, execution, checkpoint, evidence, and verified 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 capability-aware scheduling.

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 capability-aware scheduling 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 capability-aware scheduling.
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 Resource and quota scheduling

Resource and quota scheduling must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Allocate CPU, memory, GPU, tokens, provider rate, tool concurrency, storage, and network budgets. 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 mission contract, fabric control plane, harness placement, context, capability grant, execution, checkpoint, evidence, and verified 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 resource and quota scheduling.
- 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

```
mission:
  id: mission-042
  state: admitted
  harnesses:
    planner:
      model_class: frontier-reasoner
      capabilities: [read:repo, write:plan]
    implementer:
      isolation: microvm
      capabilities: [read:repo, write:worktree, execute:test]
    verifier:
      isolation: separate-image
      capabilities: [read:candidate, execute:hidden-tests]
  fan_in:
    requires: [signed-plan, candidate-patch, test-evidence, independent-review]
  production_effects:
    default: deny
```

Failure patterns

- Treating resource and quota scheduling 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 resource and quota scheduling.
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 Priority, fairness, and preemption

Priority, fairness, and preemption must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Protect interactive work, critical missions, tenant fairness, deadlines, and resumable background work. 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 mission contract, fabric control plane, harness placement, context, capability grant, execution, checkpoint, evidence, and verified 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 priority, fairness, and preemption.
- 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 priority, fairness, and preemption 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 priority, fairness, and preemption.
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 Locality and sovereign placement

Locality and sovereign placement must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Keep sensitive data, models, artifacts, or tools inside approved devices, sites, regions, or trust domains. 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 mission contract, fabric control plane, harness placement, context, capability grant, execution, checkpoint, evidence, and verified 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 locality and sovereign placement.

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 locality and sovereign placement 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 locality and sovereign placement.
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

Build a scheduler that routes ten tasks across local, cloud, GPU, restricted, and low-cost harness pools under quotas.

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 market-like model and harness scheduling constrained by sovereignty, risk, energy, and evidence quality.

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

Failure containment, recovery, and compensation

Keep one faulty worker or tool from corrupting the mission or surrounding systems.

Chapter operating position

Keep one faulty worker or tool from corrupting the mission or surrounding systems.



7.1 Harness and worker failure

Harness and worker failure must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Detect heartbeat loss, deadlock, resource exhaustion, corrupt state, model failure, and sandbox crash. 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 mission contract, fabric control plane, harness placement, context, capability grant, execution, checkpoint, evidence, and verified 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 harness and worker failure.

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 harness and worker failure 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 harness and worker failure.
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 Tool and dependency failure

Tool and dependency failure must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Handle timeouts, partial effects, duplicate responses, stale data, rate limits, and unavailable 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 mission contract, fabric control plane, harness placement, context, capability grant, execution, checkpoint, evidence, and verified 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 tool and dependency failure.
- 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

```
mission:
  id: mission-042
  state: admitted
  harnesses:
    planner:
      model_class: frontier-reasoner
      capabilities: [read:repo, write:plan]
    implementer:
      isolation: microvm
      capabilities: [read:repo, write:worktree, execute:test]
    verifier:
      isolation: separate-image
      capabilities: [read:candidate, execute:hidden-tests]
  fan_in:
    requires: [signed-plan, candidate-patch, test-evidence, independent-review]
  production_effects:
    default: deny
```

Failure patterns

- Treating tool and dependency failure 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 tool and dependency failure.
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 Compensation and rollback

Compensation and rollback must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Reverse file, repository, cloud, deployment, identity, and workflow effects through explicit compensators. 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 mission contract, fabric control plane, harness placement, context, capability grant, execution, checkpoint, evidence, and verified 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 compensation and rollback.
- 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 compensation and rollback 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 compensation and rollback.
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 Quarantine and incident response

Quarantine and incident response must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Freeze artifacts, revoke credentials, preserve evidence, isolate versions, and rebuild trusted state. 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 mission contract, fabric control plane, harness placement, context, capability grant, execution, checkpoint, evidence, and verified 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 quarantine and incident response.

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 quarantine and incident response 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 quarantine and incident response.
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

Inject worker loss, duplicate completion, partial cloud mutation, and corrupted checkpoint into one mission and recover safely.

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 self-healing harness pools with independent recovery agents and immutable 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 08

Fabric assurance, observability, and evolution

Measure the fabric as a safety-critical distributed execution platform.

Chapter operating position

Measure the fabric as a safety-critical distributed execution platform.



8.1 Trace and evidence model

Trace and evidence model must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Correlate mission, task, harness, model, context, tool, artifact, approval, cost, and outcome. 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 mission contract, fabric control plane, harness placement, context, capability grant, execution, checkpoint, evidence, and verified 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 trace and evidence model.

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 trace and evidence model 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 trace and evidence model.
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 Evaluation and chaos testing

Evaluation and chaos testing must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Test permissions, isolation, long horizons, race conditions, worker loss, stale state, and malicious inputs. 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 mission contract, fabric control plane, harness placement, context, capability grant, execution, checkpoint, evidence, and verified 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 evaluation and chaos testing.

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

```
mission:
  id: mission-042
  state: admitted
  harnesses:
    planner:
      model_class: frontier-reasoner
      capabilities: [read:repo, write:plan]
    implementer:
      isolation: microvm
      capabilities: [read:repo, write:worktree, execute:test]
    verifier:
      isolation: separate-image
      capabilities: [read:candidate, execute:hidden-tests]
  fan_in:
    requires: [signed-plan, candidate-patch, test-evidence, independent-review]
  production_effects:
    default: deny
```

Failure patterns

- Treating evaluation and chaos testing 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 evaluation and chaos testing.
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 Release and compatibility

Release and compatibility must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Version harness contracts, adapters, images, tools, schemas, policies, models, and migration 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 mission contract, fabric control plane, harness placement, context, capability grant, execution, checkpoint, evidence, and verified 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 release and compatibility.
- 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 release and compatibility 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 release and compatibility.
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 Frontier governance

Frontier governance must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Separate stable fabric primitives from experimental autonomy, protocols, schedulers, and self-modification. 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 mission contract, fabric control plane, harness placement, context, capability grant, execution, checkpoint, evidence, and verified 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 frontier governance.

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 frontier governance 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 frontier governance.
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 release qualification suite for a harness runtime and execute isolation, replay, failover, and prohibited-action tests.

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

Develop a multi-harness operating system with typed effects, federated identity, and continuous assurance.

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. Model Context Protocol - Model Context Protocol Specification 2025-11-25

Role: Stable protocol baseline for tool, resource, prompt, transport, authorization, lifecycle, and capability integration.

Verified: 2026-07-26

Official source: <https://modelcontextprotocol.io/specification/2025-11-25>

02. Model Context Protocol - The 2026 MCP Roadmap

Role: Current roadmap for transport scalability, agent communication, governance maturation, and enterprise readiness.

Verified: 2026-07-26

Official source: <https://blog.modelcontextprotocol.io/posts/2026-mcp-roadmap/>

03. Agent2Agent Protocol - Agent2Agent Protocol Specification 1.0

Role: Open protocol for agent discovery, communication, task delegation, status, artifacts, and interoperability.

Verified: 2026-07-26

Official source: <https://a2a-protocol.org/v1.0.0/>

04. NIST - Artificial Intelligence Risk Management Framework 1.0

Role: Govern, Map, Measure, and Manage framework for trustworthy AI risk.

Verified: 2026-07-26

Official source: <https://nvlpubs.nist.gov/nistpubs/ai/nist.ai.100-1.pdf>

05. NIST - NIST AI 600-1 - Generative Artificial Intelligence Profile

Role: Generative-AI risk profile and recommended actions.

Verified: 2026-07-26

Official source: <https://nvlpubs.nist.gov/nistpubs/ai/NIST.AI.600-1.pdf>

06. OWASP GenAI Security Project - Top 10 for Agentic Applications

Role: Agentic-system risks involving goals, tools, identity, memory, orchestration, and autonomy.

Verified: 2026-07-26

Official source: <https://genai.owasp.org/>

07. SPIFFE - The SPIFFE Standard

Role: Workload identities, SVIDs, trust domains, federation, and Workload API.

Verified: 2026-07-26

Official source: <https://spiffe.io/docs/latest/spiffe-specs/>

08. Ray Project - Ray Core Documentation

Role: Distributed tasks, actors, placement, scheduling, resources, and fault tolerance.

Verified: 2026-07-26

Official source: <https://docs.ray.io/en/latest/ray-core/walkthrough.html>

09. Temporal - Temporal Documentation

Role: Durable workflow execution, retries, timers, signals, state, and recovery.

Verified: 2026-07-26

Official source: <https://docs.temporal.io/>

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-AI - Artificial Intelligence & Agentic Systems
Secondary domain	MYTHOS-SEC; MYTHOS-SWE; MYTHOS-CLD
Publication class	Field Guide / Canonical Living Overview Guide
Skill levels	L1 Foundational; L2 Practitioner; L3 Advanced; L4 Frontier
Tags	agent harness, multi-harness fabric, runtime isolation, sandbox, workload identity, mission state, event sourcing, capability security, agent runtime, governance
Canonical route	/library/field-guides/mythos-fg-ai-0010/

Living publication

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