



CANONICAL LIVING OVERVIEW GUIDE

Multi-Agent and Multi-Model Parallelism

A frontier guide to extreme parallel agent systems: task graphs, role arrays, model portfolios, speculative branches, supervisors, distributed scheduling, context partitioning, consensus, verification, failure isolation, budgets, and scalable fan-in.

PERMANENT PUBLICATION IDENTITY

Multi-Agent and Multi-Model Parallelism

Document ID

MYTHOS-FG-AI-0011

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-CLD; MYTHOS-DAT; MYTHOS-SWE
Audience	Agent architects, distributed-systems engineers, AI platform teams, model-routing engineers, and advanced software-development organizations.
Prerequisites	Agent runtimes, distributed systems, evaluation, model capabilities, queues, concurrency, and observability.
Publisher / owner	Mythos Systems / Aaron Stovall
Public classification	Public technical guidance

Expected outcomes

- Decompose complex missions into parallelizable, dependency-aware, verifiable work.
- Route roles and tasks across heterogeneous local and cloud models according to capability, cost, risk, and context.
- Coordinate hundreds of workers through queues, leases, backpressure, quotas, and deterministic fan-in.
- Prevent parallelism from multiplying duplicated work, correlated errors, context leakage, and unverified claims.
- Measure useful parallel efficiency, evidence diversity, solution quality, reliability, and total system cost.

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 - Parallelism taxonomy and mission graphs	5
Chapter 01 laboratory and review	10
Chapter 02 - Agent roles, teams, and supervisory topology	11
Chapter 02 laboratory and review	16
Chapter 03 - Multi-model portfolio and routing	17
Chapter 03 laboratory and review	22
Chapter 04 - Distributed scheduling and extreme scale	23
Chapter 04 laboratory and review	28
Chapter 05 - Context partitioning and shared state	29
Chapter 05 laboratory and review	34

Chapter 06 - Consensus, fan-in, and verification	35
Chapter 06 laboratory and review	40
Chapter 07 - Reliability, failure domains, and recovery	41
Chapter 07 laboratory and review	46
Chapter 08 - Parallel efficiency, economics, and frontier scaling	47
Chapter 08 laboratory and review	52
Integrated learning roadmap	53
Source authority register	54
Limitations, freshness, and revision control	55

CHAPTER 01

Parallelism taxonomy and mission graphs

Identify where parallel work creates genuine independence and where coordination dominates.

Chapter operating position

Identify where parallel work creates genuine independence and where coordination dominates.

1.1 Task DAGs and dependency analysis

Task DAGs and dependency analysis must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Separate prerequisites, independent branches, barriers, critical paths, optional work, and validation gates. 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 DAG, task admission, role and model route, worker execution, shared state, fan-in, verification, cost, and completion 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 task dags and dependency analysis.

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 task dags and dependency analysis 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 task dags and dependency analysis.
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 Fan-out and fan-in

Fan-out and fan-in must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Dispatch bounded subtasks and reconcile typed results through deterministic aggregation. 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 DAG, task admission, role and model route, worker execution, shared state, fan-in, verification, cost, and completion 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 fan-out and fan-in.

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

```
parallel_policy:
  max_workers: 64
  max_model_calls: 400
  per_model_concurrency:
    local-fast: 24
    cloud-reasoner: 8
    verifier: 12
  queues:
    planning: {priority: 100, max_depth: 50}
    implementation: {priority: 50, max_depth: 200}
    verification: {priority: 90, max_depth: 100}
  stop_when:
    - critical_path_complete
    - marginal_verified_value_below: 0.05
    - budget_exhausted
```

Failure patterns

- Treating fan-out and fan-in 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 fan-out and fan-in.
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 Pipeline and streaming parallelism

Pipeline and streaming parallelism must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Overlap research, implementation, tests, simulation, review, and deployment 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 DAG, task admission, role and model route, worker execution, shared state, fan-in, verification, cost, and completion 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 pipeline and streaming parallelism.
- 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 pipeline and streaming parallelism 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 pipeline and streaming parallelism.
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 Speculative and search parallelism

Speculative and search parallelism must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Explore alternative plans, patches, hypotheses, tool paths, and models before selecting. 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 DAG, task admission, role and model route, worker execution, shared state, fan-in, verification, cost, and completion 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 speculative and search parallelism.

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 speculative and search parallelism 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 speculative and search parallelism.
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

Convert a monolithic engineering task into a DAG and compare serial, fan-out, pipeline, and speculative execution.

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 automatically learned mission graphs constrained by formal dependencies.

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

Agent roles, teams, and supervisory topology

Design organizational structure for machine workers rather than assigning arbitrary personas.

Chapter operating position

Design organizational structure for machine workers rather than assigning arbitrary personas.



2.1 Supervisor and worker hierarchy

Supervisor and worker hierarchy must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Assign decomposition, scheduling, conflict resolution, verification, escalation, and final decision authority. 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 DAG, task admission, role and model route, worker execution, shared state, fan-in, verification, cost, and completion 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 supervisor and worker hierarchy.

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 supervisor and worker hierarchy 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 supervisor and worker hierarchy.
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 Specialist arrays

Specialist arrays must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Use architecture, security, implementation, testing, documentation, operations, and domain specialists. 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 DAG, task admission, role and model route, worker execution, shared state, fan-in, verification, cost, and completion 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 specialist arrays.

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

```
parallel_policy:
  max_workers: 64
  max_model_calls: 400
  per_model_concurrency:
    local-fast: 24
    cloud-reasoner: 8
    verifier: 12
  queues:
    planning: {priority: 100, max_depth: 50}
    implementation: {priority: 50, max_depth: 200}
    verification: {priority: 90, max_depth: 100}
  stop_when:
    - critical_path_complete
    - marginal_verified_value_below: 0.05
    - budget_exhausted
```

Failure patterns

Treating specialist arrays 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 specialist arrays.
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 Peer review and debate

Peer review and debate must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Generate independent analyses, adversarial critiques, counterexamples, and reconciled conclusions. 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 DAG, task admission, role and model route, worker execution, shared state, fan-in, verification, cost, and completion 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 peer review and debate.

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 peer review and debate 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 peer review and debate.
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 Dynamic team formation

Dynamic team formation must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Instantiate or retire roles according to uncertainty, failures, workload, and evidence gaps. 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 DAG, task admission, role and model route, worker execution, shared state, fan-in, verification, cost, and completion 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 dynamic team formation.

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 dynamic team formation 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 dynamic team formation.
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

Run a supervisor with four specialists and compare majority vote, evidence-weighted review, and deterministic verification.

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-organizing teams with bounded authority and measurable role utility.

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

Multi-model portfolio and routing

Use model heterogeneity as an engineering resource rather than a branding choice.

Chapter operating position

Use model heterogeneity as an engineering resource rather than a branding choice.



3.1 Capability profiles

Capability profiles must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Measure reasoning, code, retrieval, tool use, context, language, modality, safety, latency, and cost. 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 DAG, task admission, role and model route, worker execution, shared state, fan-in, verification, cost, and completion 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 profiles.
- 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 profiles 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 profiles.
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 Role-to-model mapping

Role-to-model mapping must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Assign strong planners, fast workers, local private models, critics, judges, and domain adapters. 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 DAG, task admission, role and model route, worker execution, shared state, fan-in, verification, cost, and completion 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 role-to-model mapping.

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

```
parallel_policy:
  max_workers: 64
  max_model_calls: 400
  per_model_concurrency:
    local-fast: 24
    cloud-reasoner: 8
    verifier: 12
  queues:
    planning: {priority: 100, max_depth: 50}
    implementation: {priority: 50, max_depth: 200}
    verification: {priority: 90, max_depth: 100}
  stop_when:
    - critical_path_complete
    - marginal_verified_value_below: 0.05
    - budget_exhausted
```

Failure patterns

Treating role-to-model mapping 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 role-to-model mapping.
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 Cascade and escalation

Cascade and escalation must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Start with low-cost or local models and escalate when confidence, complexity, or verification demands it. 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 DAG, task admission, role and model route, worker execution, shared state, fan-in, verification, cost, and completion 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 cascade and escalation.

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 cascade and escalation 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 cascade and escalation.
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 Ensemble diversity and correlation

Ensemble diversity and correlation must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Measure whether model families produce independent evidence or repeat the same failure. 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 DAG, task admission, role and model route, worker execution, shared state, fan-in, verification, cost, and completion 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 ensemble diversity and correlation.

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 ensemble diversity and correlation 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 ensemble diversity and correlation.
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

Create a routing matrix for five models and evaluate static roles, cascades, and adaptive selection on hidden tasks.

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 online model routing using calibrated capability, uncertainty, current load, and task embeddings.

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

Distributed scheduling and extreme scale

Control large worker populations without overload or starvation.

Chapter operating position

Control large worker populations without overload or starvation.

4.1 Queues, leases, and work stealing

Queues, leases, and work stealing must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Use durable admission, task claims, heartbeats, expiry, retries, locality, and duplicate suppression. 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 DAG, task admission, role and model route, worker execution, shared state, fan-in, verification, cost, and completion 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 queues, leases, and work stealing.

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 queues, leases, and work stealing 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 queues, leases, and work stealing.
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 Resource-aware placement

Resource-aware placement must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Allocate CPU, GPU, memory, context, tokens, rate limits, tools, and sovereign data access. The design should identify authoritative state, workload identities, typed contracts, resource and token budgets, trust boundaries, failure domains, telemetry, rollback, and acceptance evidence.

This guide traces the subject through the mission DAG, task admission, role and model route, worker execution, shared state, fan-in, verification, cost, and completion 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-aware 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.

Implementation sketch

```
parallel_policy:
  max_workers: 64
  max_model_calls: 400
  per_model_concurrency:
    local-fast: 24
    cloud-reasoner: 8
    verifier: 12
  queues:
    planning: {priority: 100, max_depth: 50}
    implementation: {priority: 50, max_depth: 200}
    verification: {priority: 90, max_depth: 100}
  stop_when:
    - critical_path_complete
    - marginal_verified_value_below: 0.05
    - budget_exhausted
```

Failure patterns

Treating resource-aware 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 resource-aware 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.

4.3 Backpressure and concurrency control

Backpressure and concurrency control must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Bound active workers, queue growth, tool contention, model saturation, and downstream fan-in. 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 DAG, task admission, role and model route, worker execution, shared state, fan-in, verification, cost, and completion 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 backpressure and concurrency control.

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 backpressure and concurrency control 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 backpressure and concurrency control.
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 Cohorts, quotas, and fairness

Cohorts, quotas, and fairness must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Share scarce models and accelerators across missions, teams, tenants, and priorities. 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 DAG, task admission, role and model route, worker execution, shared state, fan-in, verification, cost, and completion 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 cohorts, quotas, and fairness.

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 cohorts, quotas, and fairness 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 cohorts, quotas, and fairness.
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

Simulate one thousand subtasks with worker loss, heterogeneous duration, limited model quotas, and a bounded fan-in service.

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 Kueue-style admission for agent swarms and cross-cluster harness placement.

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

Context partitioning and shared state

Provide each worker enough information without cloning the entire mission history.

Chapter operating position

Provide each worker enough information without cloning the entire mission history.

5.1 Role-specific context views

Role-specific context views must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Project task contract, authoritative sources, dependencies, local state, and acceptance criteria. 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 DAG, task admission, role and model route, worker execution, shared state, fan-in, verification, cost, and completion 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 role-specific context views.

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 role-specific context views 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 role-specific context views.
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 Blackboard and shared memory

Blackboard and shared memory must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Publish claims, artifacts, questions, conflicts, evidence, and status through typed records. 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 DAG, task admission, role and model route, worker execution, shared state, fan-in, verification, cost, and completion 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 blackboard and shared memory.

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

```
parallel_policy:
  max_workers: 64
  max_model_calls: 400
  per_model_concurrency:
    local-fast: 24
    cloud-reasoner: 8
    verifier: 12
  queues:
    planning: {priority: 100, max_depth: 50}
    implementation: {priority: 50, max_depth: 200}
    verification: {priority: 90, max_depth: 100}
  stop_when:
    - critical_path_complete
    - marginal_verified_value_below: 0.05
    - budget_exhausted
```

Failure patterns

Treating blackboard and shared memory 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 blackboard and shared memory.
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 Private scratch and confidentiality

Private scratch and confidentiality must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Separate local reasoning, restricted data, tenant context, and sensitive tool results. 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 DAG, task admission, role and model route, worker execution, shared state, fan-in, verification, cost, and completion 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 private scratch and confidentiality.

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 private scratch and confidentiality 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 private scratch and confidentiality.
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 Freshness and concurrency

Freshness and concurrency must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Detect stale plans, changed artifacts, conflicting edits, duplicated work, and inconsistent summaries. 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 DAG, task admission, role and model route, worker execution, shared state, fan-in, verification, cost, and completion 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 freshness and concurrency.

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 freshness and concurrency 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 freshness and concurrency.
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

Partition a repository mission among ten workers and prove that no worker receives unrelated secrets or stale dependencies.

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 semantic memory and cryptographic access control for shared agent state.

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

Consensus, fan-in, and verification

Turn many outputs into one justified result without averaging errors.

Chapter operating position

Turn many outputs into one justified result without averaging errors.

6.1 Typed result contracts

Typed result contracts must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Require claims, evidence, artifacts, uncertainty, assumptions, tests, and unresolved issues. 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 DAG, task admission, role and model route, worker execution, shared state, fan-in, verification, cost, and completion 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 result contracts.

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

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

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

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

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

Failure patterns

Treating typed result contracts as a feature rather than an end-to-end stateful system.

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

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

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

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

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

Validation and evidence

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

6.2 Deterministic aggregation

Deterministic aggregation must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Merge code, data, reports, and state through schemas, compilers, tests, and transaction rules. 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 DAG, task admission, role and model route, worker execution, shared state, fan-in, verification, cost, and completion 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 deterministic aggregation.

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

```
parallel_policy:
  max_workers: 64
  max_model_calls: 400
  per_model_concurrency:
    local-fast: 24
    cloud-reasoner: 8
    verifier: 12
  queues:
    planning: {priority: 100, max_depth: 50}
    implementation: {priority: 50, max_depth: 200}
    verification: {priority: 90, max_depth: 100}
  stop_when:
    - critical_path_complete
    - marginal_verified_value_below: 0.05
    - budget_exhausted
```

Failure patterns

Treating deterministic aggregation 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 deterministic aggregation.
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 Evidence-weighted consensus

Evidence-weighted consensus must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Prefer source authority, independent replication, stronger tests, and calibrated expertise over votes. 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 DAG, task admission, role and model route, worker execution, shared state, fan-in, verification, cost, and completion chain. A running agent, reconciled object, successful API call, healthy proxy, or completed function is not proof of the intended outcome. Desired state must be compared with applied state, external effects, negative tests, degraded behavior, causal evidence, cost, and user or mission results.

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

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

Engineering practices

Define authority, ownership, desired state, dependencies, limits, and evidence for evidence-weighted consensus.

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

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

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

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

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

Failure patterns

Treating evidence-weighted consensus as a feature rather than an end-to-end stateful system.

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

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

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

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

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

Validation and evidence

Method	Expected evidence
Examine	Requirements, APIs, identities, state machines, policies, versions, resources, and dependencies for evidence-weighted consensus.
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 Conflict and minority findings

Conflict and minority findings must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Preserve dissent, reproduce disagreements, request targeted evidence, and escalate high-impact uncertainty. 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 DAG, task admission, role and model route, worker execution, shared state, fan-in, verification, cost, and completion 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 conflict and minority findings.

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 conflict and minority findings 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 conflict and minority findings.
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

Aggregate twenty agent reports containing duplicates, conflicts, unsupported claims, and one critical minority finding.

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 formal proof aggregation and verifier networks.

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

Reliability, failure domains, and recovery

Prevent correlated errors and cascading retries from overwhelming the system.

Chapter operating position

Prevent correlated errors and cascading retries from overwhelming the system.

7.1 Worker and supervisor failure

Worker and supervisor failure must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Recover leases, restart tasks, promote backups, preserve state, and avoid split-brain leadership. 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 DAG, task admission, role and model route, worker execution, shared state, fan-in, verification, cost, and completion 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 worker and supervisor 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 worker and supervisor 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 worker and supervisor 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 Correlated model failure

Correlated model failure must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Detect shared training artifacts, prompt bias, benchmark shortcuts, and synchronized hallucination. 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 DAG, task admission, role and model route, worker execution, shared state, fan-in, verification, cost, and completion 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 correlated model 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

```
parallel_policy:
  max_workers: 64
  max_model_calls: 400
  per_model_concurrency:
    local-fast: 24
    cloud-reasoner: 8
    verifier: 12
  queues:
    planning: {priority: 100, max_depth: 50}
    implementation: {priority: 50, max_depth: 200}
    verification: {priority: 90, max_depth: 100}
  stop_when:
    - critical_path_complete
    - marginal_verified_value_below: 0.05
    - budget_exhausted
```

Failure patterns

Treating correlated model 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 correlated model 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 Retry and speculation storms

Retry and speculation storms must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Use budgets, jitter, circuit breakers, deduplication, and global overload controls. 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 DAG, task admission, role and model route, worker execution, shared state, fan-in, verification, cost, and completion 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 retry and speculation storms.

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 retry and speculation storms 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 retry and speculation storms.
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 Mission checkpoint and compensation

Mission checkpoint and compensation must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Resume after fabric outage, roll back external effects, and reconcile orphan artifacts. 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 DAG, task admission, role and model route, worker execution, shared state, fan-in, verification, cost, and completion 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 checkpoint and compensation.

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 checkpoint and compensation 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 checkpoint and compensation.
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 supervisor loss, a shared-model systematic error, and a retry storm into a parallel mission.

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 Byzantine-resilient agent collaboration for high-assurance environments.

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

Parallel efficiency, economics, and frontier scaling

Measure useful work rather than raw agent count.

Chapter operating position

Measure useful work rather than raw agent count.

8.1 Speedup and critical path

Speedup and critical path must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Measure wall time, serial fraction, queueing, synchronization, fan-in, and rework. 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 DAG, task admission, role and model route, worker execution, shared state, fan-in, verification, cost, and completion 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 speedup and critical path.

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 speedup and critical path 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 speedup and critical path.
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 Quality and evidence gain

Quality and evidence gain must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Measure solved tasks, critical failures, independent confirmations, coverage, and verified novelty. 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 DAG, task admission, role and model route, worker execution, shared state, fan-in, verification, cost, and completion 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 quality and evidence gain.

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

```
parallel_policy:
  max_workers: 64
  max_model_calls: 400
  per_model_concurrency:
    local-fast: 24
    cloud-reasoner: 8
    verifier: 12
  queues:
    planning: {priority: 100, max_depth: 50}
    implementation: {priority: 50, max_depth: 200}
    verification: {priority: 90, max_depth: 100}
  stop_when:
    - critical_path_complete
    - marginal_verified_value_below: 0.05
    - budget_exhausted
```

Failure patterns

Treating quality and evidence gain 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 quality and evidence gain.
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 Cost and resource efficiency

Cost and resource efficiency must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Track tokens, model calls, GPUs, tools, storage, networking, human review, and failed branches. 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 DAG, task admission, role and model route, worker execution, shared state, fan-in, verification, cost, and completion 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 cost and resource efficiency.

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 cost and resource efficiency 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 cost and resource efficiency.
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 Adaptive scale policy

Adaptive scale policy must be engineered as a governed distributed-system mechanism, not a product label, agent persona, YAML object, or diagram. Increase or reduce agents according to uncertainty, marginal value, deadline, budget, and system health. 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 DAG, task admission, role and model route, worker execution, shared state, fan-in, verification, cost, and completion 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 adaptive scale 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 adaptive scale 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 adaptive scale 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.

Chapter 08 laboratory and review

Practical laboratory

Benchmark 1, 4, 16, and 64-agent configurations and report useful speedup, quality, cost, duplication, and fan-in burden.

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 elastic multi-model supercomputing for reasoning with policy-constrained search and evidence-aware stopping.

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. 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/
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. 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
04. Kubernetes - Kueue Documentation Role: Kubernetes-native batch and AI workload queueing, admission, cohorts, and resource sharing. Verified: 2026-07-26 Official source: https://kueue.sigs.k8s.io/
05. Temporal - Temporal Documentation Role: Durable workflow execution, retries, timers, signals, state, and recovery. Verified: 2026-07-26 Official source: https://docs.temporal.io/
06. 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
07. 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
08. OpenTelemetry - Generative AI Observability with OpenTelemetry Role: Current GenAI observability guidance for model calls, tokens, content controls, tool calls, and results. Verified: 2026-07-26 Official source: https://opentelemetry.io/blog/2026/genai-observability/

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-CLD; MYTHOS-DAT; MYTHOS-SWE
Publication class	Field Guide / Canonical Living Overview Guide
Skill levels	L1 Foundational; L2 Practitioner; L3 Advanced; L4 Frontier
Tags	multi-agent, multi-model, parallelism, agent swarm, model routing, fan-out fan-in, distributed scheduling, speculative execution, supervisor, extreme scale
Canonical route	/library/field-guides/mythos-fg-ai-0011/

Living publication

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