



CANONICAL LIVING OVERVIEW GUIDE

AI-Assisted Coding, Review, Testing, and Repository Workflows

An implementation-grade guide to using coding models and agents across requirements, planning, repository understanding, implementation, review, testing, debugging, documentation, release, provenance, and verified completion.

PERMANENT PUBLICATION IDENTITY

AI-Assisted Coding, Review, Testing, and Repository Workflows

Document ID
MYTHOS-FG-SWE-0020

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-SWE - Software Engineering & Design
Secondary domain	MYTHOS-AI; MYTHOS-SEC; MYTHOS-DAT
Audience	Software engineers, AI-assisted developers, technical leads, reviewers, platform teams, and agentic-development architects.
Prerequisites	Software requirements, architecture, source control, testing, CI/CD, security, and model fundamentals.
Publisher / owner	Mythos Systems / Aaron Stovall
Public classification	Public technical guidance

Expected outcomes

- Design AI-assisted workflows around explicit specifications, authority, evidence, and acceptance criteria.
- Provide models with repository context without uncontrolled disclosure or context waste.
- Use single-agent and multi-agent patterns for planning, coding, review, tests, debugging, and documentation.
- Evaluate generated changes through deterministic checks, adversarial review, provenance, and rollback.
- Operate coding agents safely across branches, worktrees, tools, credentials, and release workflows.

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 - Operating model for AI-assisted development	5
Chapter 01 laboratory and review	10
Chapter 02 - Specifications, plans, and acceptance contracts	11
Chapter 02 laboratory and review	16
Chapter 03 - Repository context and codebase understanding	17
Chapter 03 laboratory and review	22
Chapter 04 - Implementation, branching, and parallel agent work	23
Chapter 04 laboratory and review	28
Chapter 05 - AI-assisted review and security assurance	29
Chapter 05 laboratory and review	34

Chapter 06 - Generated tests, debugging, and diagnosis	35
Chapter 06 laboratory and review	40
Chapter 07 - CI/CD, provenance, and verified completion	41
Chapter 07 laboratory and review	46
Chapter 08 - Economics, governance, and continuous improvement	47
Chapter 08 laboratory and review	52
Integrated learning roadmap	53
Source authority register	54
Limitations, freshness, and revision control	55

CHAPTER 01

Operating model for AI-assisted development

Define model roles, human authority, and the software lifecycle boundary.

Chapter operating position

Define model roles, human authority, and the software lifecycle boundary.



1.1 Capability and risk tiers

Capability and risk tiers must be treated as an observable AI-system mechanism rather than a model label, benchmark claim, prompt trick, or framework feature. Separate autocomplete, explanation, patch generation, autonomous tasks, and release actions by authority and blast radius. The implementation should name authoritative inputs, data and model versions, identities, state transitions, permissions, resource limits, telemetry, failure behavior, uncertainty, rollback, and acceptance criteria.

This guide traces the subject through the request, specification, repository context, plan, patch, test, review, evidence, release, and user outcome chain. A fluent response, high aggregate score, successful tool call, or green service dashboard is not sufficient proof. Intended behavior must be reconciled with hidden tests, source evidence, negative and adversarial cases, latency and resource measurements, user-visible results, and residual limitations.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one model, dataset, retrieval source, tool, memory tier, optimization, or permission at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated plans and model judgments remain subject to deterministic validation and human authority.

Troubleshooting locates the first divergence from the expected contract or state machine. Account for tokenizer and template mismatch, stale data, context loss, model nondeterminism, caching, quantization, scheduling, permissions, prompt injection, missing telemetry, partial tool effects, and distribution shift. Completion requires reproducibility, causal evidence, validated recovery, and clear disclosure of uncertainty.

Engineering practices

Define the objective, authority, data, model, owner, constraints, risks, and acceptance criteria for capability and risk tiers.

Separate model behavior from retrieval, tools, memory, policy, interface, and human oversight.

Version prompts, schemas, data, model artifacts, runtime configuration, evaluations, and release evidence.

Test nominal, negative, adversarial, long-context, degraded, overloaded, recovery, and rollback conditions.

Retain source, trajectory, tool, citation, metric, profile, decision, and user-outcome evidence.

Prefer least privilege, typed contracts, bounded budgets, staged rollout, abstention, and reversibility.

Failure patterns

Treating capability and risk tiers as a model capability claim disconnected from complete system behavior.

Using one benchmark, model judge, or successful demonstration as proof of production readiness.

Ignoring tokenizer, prompt template, source authority, permissions, cache, hardware, or version boundaries.

Allowing tools, plugins, retrieval content, or memory to redefine trusted instructions.

Optimizing latency, tokens, or cost without measuring quality, safety, and tail failures.

Declaring completion without hidden tests, evidence, residual limitations, rollback, and monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, authority, model, data, prompts, tools, memory, schemas, versions, and assumptions for capability and risk tiers.
Observe	Requests, selected context, trajectories, tool calls, model outputs, citations, caches, telemetry, resources, and user outcomes.
Test	Expected, prohibited, adversarial, malformed, long-context, overloaded, failed, recovery, and rollback cases.
Measure	Task quality, calibration, grounding, safety, latency, throughput, memory, tokens, cost, energy, and operator effort.
Reconcile	Intended system behavior against evaluated model, data, runtime, tools, sources, deployment, and residual risk.

1.2 Human and model responsibilities

Human and model responsibilities must be treated as an observable AI-system mechanism rather than a model label, benchmark claim, prompt trick, or framework feature. Assign requirements, architecture, implementation, review, approval, deployment, and incident authority. The implementation should name authoritative inputs, data and model versions, identities, state transitions, permissions, resource limits, telemetry, failure behavior, uncertainty, rollback, and acceptance criteria.

This guide traces the subject through the request, specification, repository context, plan, patch, test, review, evidence, release, and user outcome chain. A fluent response, high aggregate score, successful tool call, or green service dashboard is not sufficient proof. Intended behavior must be reconciled with hidden tests, source evidence, negative and adversarial cases, latency and resource measurements, user-visible results, and residual limitations.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one model, dataset, retrieval source, tool, memory tier, optimization, or permission at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated plans and model judgments remain subject to deterministic validation and human authority.

Troubleshooting locates the first divergence from the expected contract or state machine. Account for tokenizer and template mismatch, stale data, context loss, model nondeterminism, caching, quantization, scheduling, permissions, prompt injection, missing telemetry, partial tool effects, and distribution shift. Completion requires reproducibility, causal evidence, validated recovery, and clear disclosure of uncertainty.

Engineering practices

Define the objective, authority, data, model, owner, constraints, risks, and acceptance criteria for human and model responsibilities.

Separate model behavior from retrieval, tools, memory, policy, interface, and human oversight.

Version prompts, schemas, data, model artifacts, runtime configuration, evaluations, and release evidence.

Test nominal, negative, adversarial, long-context, degraded, overloaded, recovery, and rollback conditions.

Retain source, trajectory, tool, citation, metric, profile, decision, and user-outcome evidence.

Prefer least privilege, typed contracts, bounded budgets, staged rollout, abstention, and reversibility.

Implementation sketch

```
task:
  objective: add tenant-safe export
  constraints: [no-new-dependency, backward-compatible-api]
  repository_context:
    authoritative: [architecture.md, src/export, tests/export]
    excluded: [secrets, customer-data]
  tracks: [implementation, tests, independent-review]
  completion:
    - hidden-tests-pass
    - cross-tenant-access-denied
    - migration-and-rollback-verified
    - provenance-recorded
```

Failure patterns

Treating human and model responsibilities as a model capability claim disconnected from complete system behavior.

Using one benchmark, model judge, or successful demonstration as proof of production readiness.

Ignoring tokenizer, prompt template, source authority, permissions, cache, hardware, or version boundaries.

Allowing tools, plugins, retrieval content, or memory to redefine trusted instructions.

Optimizing latency, tokens, or cost without measuring quality, safety, and tail failures.

Declaring completion without hidden tests, evidence, residual limitations, rollback, and monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, authority, model, data, prompts, tools, memory, schemas, versions, and assumptions for human and model responsibilities.
Observe	Requests, selected context, trajectories, tool calls, model outputs, citations, caches, telemetry, resources, and user outcomes.
Test	Expected, prohibited, adversarial, malformed, long-context, overloaded, failed, recovery, and rollback cases.
Measure	Task quality, calibration, grounding, safety, latency, throughput, memory, tokens, cost, energy, and operator effort.
Reconcile	Intended system behavior against evaluated model, data, runtime, tools, sources, deployment, and residual risk.



1.3 Workflow states and artifacts

Workflow states and artifacts must be treated as an observable AI-system mechanism rather than a model label, benchmark claim, prompt trick, or framework feature. Use request, specification, plan, patch, tests, evidence, review, approval, merge, release, and rollback states. The implementation should name authoritative inputs, data and model versions, identities, state transitions, permissions, resource limits, telemetry, failure behavior, uncertainty, rollback, and acceptance criteria.

This guide traces the subject through the request, specification, repository context, plan, patch, test, review, evidence, release, and user outcome chain. A fluent response, high aggregate score, successful tool call, or green service dashboard is not sufficient proof. Intended behavior must be reconciled with hidden tests, source evidence, negative and adversarial cases, latency and resource measurements, user-visible results, and residual limitations.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one model, dataset, retrieval source, tool, memory tier, optimization, or permission at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated plans and model judgments remain subject to deterministic validation and human authority.

Troubleshooting locates the first divergence from the expected contract or state machine. Account for tokenizer and template mismatch, stale data, context loss, model nondeterminism, caching, quantization, scheduling, permissions, prompt injection, missing telemetry, partial tool effects, and distribution shift. Completion requires reproducibility, causal evidence, validated recovery, and clear disclosure of uncertainty.

Engineering practices

Define the objective, authority, data, model, owner, constraints, risks, and acceptance criteria for workflow states and artifacts.

Separate model behavior from retrieval, tools, memory, policy, interface, and human oversight.

Version prompts, schemas, data, model artifacts, runtime configuration, evaluations, and release evidence.

Test nominal, negative, adversarial, long-context, degraded, overloaded, recovery, and rollback conditions.

Retain source, trajectory, tool, citation, metric, profile, decision, and user-outcome evidence.

Prefer least privilege, typed contracts, bounded budgets, staged rollout, abstention, and reversibility.

Failure patterns

Treating workflow states and artifacts as a model capability claim disconnected from complete system behavior.

Using one benchmark, model judge, or successful demonstration as proof of production readiness.

Ignoring tokenizer, prompt template, source authority, permissions, cache, hardware, or version boundaries.

Allowing tools, plugins, retrieval content, or memory to redefine trusted instructions.

Optimizing latency, tokens, or cost without measuring quality, safety, and tail failures.

Declaring completion without hidden tests, evidence, residual limitations, rollback, and monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, authority, model, data, prompts, tools, memory, schemas, versions, and assumptions for workflow states and artifacts.
Observe	Requests, selected context, trajectories, tool calls, model outputs, citations, caches, telemetry, resources, and user outcomes.
Test	Expected, prohibited, adversarial, malformed, long-context, overloaded, failed, recovery, and rollback cases.
Measure	Task quality, calibration, grounding, safety, latency, throughput, memory, tokens, cost, energy, and operator effort.
Reconcile	Intended system behavior against evaluated model, data, runtime, tools, sources, deployment, and residual risk.

1.4 Provider and model portfolio

Provider and model portfolio must be treated as an observable AI-system mechanism rather than a model label, benchmark claim, prompt trick, or framework feature. Route work by capability, context, privacy, latency, cost, licensing, and local or cloud constraints. The implementation should name authoritative inputs, data and model versions, identities, state transitions, permissions, resource limits, telemetry, failure behavior, uncertainty, rollback, and acceptance criteria.

This guide traces the subject through the request, specification, repository context, plan, patch, test, review, evidence, release, and user outcome chain. A fluent response, high aggregate score, successful tool call, or green service dashboard is not sufficient proof. Intended behavior must be reconciled with hidden tests, source evidence, negative and adversarial cases, latency and resource measurements, user-visible results, and residual limitations.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one model, dataset, retrieval source, tool, memory tier, optimization, or permission at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated plans and model judgments remain subject to deterministic validation and human authority.

Troubleshooting locates the first divergence from the expected contract or state machine. Account for tokenizer and template mismatch, stale data, context loss, model nondeterminism, caching, quantization, scheduling, permissions, prompt injection, missing telemetry, partial tool effects, and distribution shift. Completion requires reproducibility, causal evidence, validated recovery, and clear disclosure of uncertainty.

Engineering practices

Define the objective, authority, data, model, owner, constraints, risks, and acceptance criteria for provider and model portfolio.

Separate model behavior from retrieval, tools, memory, policy, interface, and human oversight.

Version prompts, schemas, data, model artifacts, runtime configuration, evaluations, and release evidence.

Test nominal, negative, adversarial, long-context, degraded, overloaded, recovery, and rollback conditions.

Retain source, trajectory, tool, citation, metric, profile, decision, and user-outcome evidence.

Prefer least privilege, typed contracts, bounded budgets, staged rollout, abstention, and reversibility.

Failure patterns

Treating provider and model portfolio as a model capability claim disconnected from complete system behavior.

Using one benchmark, model judge, or successful demonstration as proof of production readiness.

Ignoring tokenizer, prompt template, source authority, permissions, cache, hardware, or version boundaries.

Allowing tools, plugins, retrieval content, or memory to redefine trusted instructions.

Optimizing latency, tokens, or cost without measuring quality, safety, and tail failures.

Declaring completion without hidden tests, evidence, residual limitations, rollback, and monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, authority, model, data, prompts, tools, memory, schemas, versions, and assumptions for provider and model portfolio.
Observe	Requests, selected context, trajectories, tool calls, model outputs, citations, caches, telemetry, resources, and user outcomes.
Test	Expected, prohibited, adversarial, malformed, long-context, overloaded, failed, recovery, and rollback cases.
Measure	Task quality, calibration, grounding, safety, latency, throughput, memory, tokens, cost, energy, and operator effort.
Reconcile	Intended system behavior against evaluated model, data, runtime, tools, sources, deployment, and residual risk.

Chapter 01 laboratory and review

Practical laboratory

Create a governed workflow for a repository change from request through release, including explicit model and human responsibilities.

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 model swarms, learned routing, verifier models, and evidence-bound autonomous development.

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

Specifications, plans, and acceptance contracts

Give models a testable definition of the requested outcome.

Chapter operating position

Give models a testable definition of the requested outcome.

2.1 Problem and stakeholder intent

Problem and stakeholder intent must be treated as an observable AI-system mechanism rather than a model label, benchmark claim, prompt trick, or framework feature. Capture user outcome, constraints, non-goals, risk, dependencies, and decision authority. The implementation should name authoritative inputs, data and model versions, identities, state transitions, permissions, resource limits, telemetry, failure behavior, uncertainty, rollback, and acceptance criteria.

This guide traces the subject through the request, specification, repository context, plan, patch, test, review, evidence, release, and user outcome chain. A fluent response, high aggregate score, successful tool call, or green service dashboard is not sufficient proof. Intended behavior must be reconciled with hidden tests, source evidence, negative and adversarial cases, latency and resource measurements, user-visible results, and residual limitations.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one model, dataset, retrieval source, tool, memory tier, optimization, or permission at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated plans and model judgments remain subject to deterministic validation and human authority.

Troubleshooting locates the first divergence from the expected contract or state machine. Account for tokenizer and template mismatch, stale data, context loss, model nondeterminism, caching, quantization, scheduling, permissions, prompt injection, missing telemetry, partial tool effects, and distribution shift. Completion requires reproducibility, causal evidence, validated recovery, and clear disclosure of uncertainty.

Engineering practices

Define the objective, authority, data, model, owner, constraints, risks, and acceptance criteria for problem and stakeholder intent.

Separate model behavior from retrieval, tools, memory, policy, interface, and human oversight.

Version prompts, schemas, data, model artifacts, runtime configuration, evaluations, and release evidence.

Test nominal, negative, adversarial, long-context, degraded, overloaded, recovery, and rollback conditions.

Retain source, trajectory, tool, citation, metric, profile, decision, and user-outcome evidence.

Prefer least privilege, typed contracts, bounded budgets, staged rollout, abstention, and reversibility.

Failure patterns

Treating problem and stakeholder intent as a model capability claim disconnected from complete system behavior.

Using one benchmark, model judge, or successful demonstration as proof of production readiness.

Ignoring tokenizer, prompt template, source authority, permissions, cache, hardware, or version boundaries.

Allowing tools, plugins, retrieval content, or memory to redefine trusted instructions.

Optimizing latency, tokens, or cost without measuring quality, safety, and tail failures.

Declaring completion without hidden tests, evidence, residual limitations, rollback, and monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, authority, model, data, prompts, tools, memory, schemas, versions, and assumptions for problem and stakeholder intent.
Observe	Requests, selected context, trajectories, tool calls, model outputs, citations, caches, telemetry, resources, and user outcomes.
Test	Expected, prohibited, adversarial, malformed, long-context, overloaded, failed, recovery, and rollback cases.
Measure	Task quality, calibration, grounding, safety, latency, throughput, memory, tokens, cost, energy, and operator effort.
Reconcile	Intended system behavior against evaluated model, data, runtime, tools, sources, deployment, and residual risk.

2.2 Architecture and change specification

Architecture and change specification must be treated as an observable AI-system mechanism rather than a model label, benchmark claim, prompt trick, or framework feature. Define affected boundaries, interfaces, data, state, compatibility, migration, and rollback. The implementation should name authoritative inputs, data and model versions, identities, state transitions, permissions, resource limits, telemetry, failure behavior, uncertainty, rollback, and acceptance criteria.

This guide traces the subject through the request, specification, repository context, plan, patch, test, review, evidence, release, and user outcome chain. A fluent response, high aggregate score, successful tool call, or green service dashboard is not sufficient proof. Intended behavior must be reconciled with hidden tests, source evidence, negative and adversarial cases, latency and resource measurements, user-visible results, and residual limitations.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one model, dataset, retrieval source, tool, memory tier, optimization, or permission at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated plans and model judgments remain subject to deterministic validation and human authority.

Troubleshooting locates the first divergence from the expected contract or state machine. Account for tokenizer and template mismatch, stale data, context loss, model nondeterminism, caching, quantization, scheduling, permissions, prompt injection, missing telemetry, partial tool effects, and distribution shift. Completion requires reproducibility, causal evidence, validated recovery, and clear disclosure of uncertainty.

Engineering practices

Define the objective, authority, data, model, owner, constraints, risks, and acceptance criteria for architecture and change specification.

Separate model behavior from retrieval, tools, memory, policy, interface, and human oversight.

Version prompts, schemas, data, model artifacts, runtime configuration, evaluations, and release evidence.

Test nominal, negative, adversarial, long-context, degraded, overloaded, recovery, and rollback conditions.

Retain source, trajectory, tool, citation, metric, profile, decision, and user-outcome evidence.

Prefer least privilege, typed contracts, bounded budgets, staged rollout, abstention, and reversibility.

Implementation sketch

```
task:
  objective: add tenant-safe export
  constraints: [no-new-dependency, backward-compatible-api]
  repository_context:
    authoritative: [architecture.md, src/export, tests/export]
    excluded: [secrets, customer-data]
  tracks: [implementation, tests, independent-review]
  completion:
    - hidden-tests-pass
    - cross-tenant-access-denied
    - migration-and-rollback-verified
    - provenance-recorded
```

Failure patterns

Treating architecture and change specification as a model capability claim disconnected from complete system behavior.

Using one benchmark, model judge, or successful demonstration as proof of production readiness.

Ignoring tokenizer, prompt template, source authority, permissions, cache, hardware, or version boundaries.

Allowing tools, plugins, retrieval content, or memory to redefine trusted instructions.

Optimizing latency, tokens, or cost without measuring quality, safety, and tail failures.

Declaring completion without hidden tests, evidence, residual limitations, rollback, and monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, authority, model, data, prompts, tools, memory, schemas, versions, and assumptions for architecture and change specification.
Observe	Requests, selected context, trajectories, tool calls, model outputs, citations, caches, telemetry, resources, and user outcomes.
Test	Expected, prohibited, adversarial, malformed, long-context, overloaded, failed, recovery, and rollback cases.
Measure	Task quality, calibration, grounding, safety, latency, throughput, memory, tokens, cost, energy, and operator effort.
Reconcile	Intended system behavior against evaluated model, data, runtime, tools, sources, deployment, and residual risk.



2.3 Task decomposition and sequencing

Task decomposition and sequencing must be treated as an observable AI-system mechanism rather than a model label, benchmark claim, prompt trick, or framework feature. Produce dependency-aware steps with owners, artifacts, checkpoints, and completion gates. The implementation should name authoritative inputs, data and model versions, identities, state transitions, permissions, resource limits, telemetry, failure behavior, uncertainty, rollback, and acceptance criteria.

This guide traces the subject through the request, specification, repository context, plan, patch, test, review, evidence, release, and user outcome chain. A fluent response, high aggregate score, successful tool call, or green service dashboard is not sufficient proof. Intended behavior must be reconciled with hidden tests, source evidence, negative and adversarial cases, latency and resource measurements, user-visible results, and residual limitations.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one model, dataset, retrieval source, tool, memory tier, optimization, or permission at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated plans and model judgments remain subject to deterministic validation and human authority.

Troubleshooting locates the first divergence from the expected contract or state machine. Account for tokenizer and template mismatch, stale data, context loss, model nondeterminism, caching, quantization, scheduling, permissions, prompt injection, missing telemetry, partial tool effects, and distribution shift. Completion requires reproducibility, causal evidence, validated recovery, and clear disclosure of uncertainty.

Engineering practices

Define the objective, authority, data, model, owner, constraints, risks, and acceptance criteria for task decomposition and sequencing.

Separate model behavior from retrieval, tools, memory, policy, interface, and human oversight.

Version prompts, schemas, data, model artifacts, runtime configuration, evaluations, and release evidence.

Test nominal, negative, adversarial, long-context, degraded, overloaded, recovery, and rollback conditions.

Retain source, trajectory, tool, citation, metric, profile, decision, and user-outcome evidence.

Prefer least privilege, typed contracts, bounded budgets, staged rollout, abstention, and reversibility.

Failure patterns

Treating task decomposition and sequencing as a model capability claim disconnected from complete system behavior.

Using one benchmark, model judge, or successful demonstration as proof of production readiness.

Ignoring tokenizer, prompt template, source authority, permissions, cache, hardware, or version boundaries.

Allowing tools, plugins, retrieval content, or memory to redefine trusted instructions.

Optimizing latency, tokens, or cost without measuring quality, safety, and tail failures.

Declaring completion without hidden tests, evidence, residual limitations, rollback, and monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, authority, model, data, prompts, tools, memory, schemas, versions, and assumptions for task decomposition and sequencing.
Observe	Requests, selected context, trajectories, tool calls, model outputs, citations, caches, telemetry, resources, and user outcomes.
Test	Expected, prohibited, adversarial, malformed, long-context, overloaded, failed, recovery, and rollback cases.
Measure	Task quality, calibration, grounding, safety, latency, throughput, memory, tokens, cost, energy, and operator effort.
Reconcile	Intended system behavior against evaluated model, data, runtime, tools, sources, deployment, and residual risk.

2.4 Acceptance and evidence

Acceptance and evidence must be treated as an observable AI-system mechanism rather than a model label, benchmark claim, prompt trick, or framework feature. Require tests, static checks, performance, security, documentation, telemetry, and user-visible proof. The implementation should name authoritative inputs, data and model versions, identities, state transitions, permissions, resource limits, telemetry, failure behavior, uncertainty, rollback, and acceptance criteria.

This guide traces the subject through the request, specification, repository context, plan, patch, test, review, evidence, release, and user outcome chain. A fluent response, high aggregate score, successful tool call, or green service dashboard is not sufficient proof. Intended behavior must be reconciled with hidden tests, source evidence, negative and adversarial cases, latency and resource measurements, user-visible results, and residual limitations.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one model, dataset, retrieval source, tool, memory tier, optimization, or permission at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated plans and model judgments remain subject to deterministic validation and human authority.

Troubleshooting locates the first divergence from the expected contract or state machine. Account for tokenizer and template mismatch, stale data, context loss, model nondeterminism, caching, quantization, scheduling, permissions, prompt injection, missing telemetry, partial tool effects, and distribution shift. Completion requires reproducibility, causal evidence, validated recovery, and clear disclosure of uncertainty.

Engineering practices

Define the objective, authority, data, model, owner, constraints, risks, and acceptance criteria for acceptance and evidence.

Separate model behavior from retrieval, tools, memory, policy, interface, and human oversight.

Version prompts, schemas, data, model artifacts, runtime configuration, evaluations, and release evidence.

Test nominal, negative, adversarial, long-context, degraded, overloaded, recovery, and rollback conditions.

Retain source, trajectory, tool, citation, metric, profile, decision, and user-outcome evidence.

Prefer least privilege, typed contracts, bounded budgets, staged rollout, abstention, and reversibility.

Failure patterns

Treating acceptance and evidence as a model capability claim disconnected from complete system behavior.

Using one benchmark, model judge, or successful demonstration as proof of production readiness.

Ignoring tokenizer, prompt template, source authority, permissions, cache, hardware, or version boundaries.

Allowing tools, plugins, retrieval content, or memory to redefine trusted instructions.

Optimizing latency, tokens, or cost without measuring quality, safety, and tail failures.

Declaring completion without hidden tests, evidence, residual limitations, rollback, and monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, authority, model, data, prompts, tools, memory, schemas, versions, and assumptions for acceptance and evidence.
Observe	Requests, selected context, trajectories, tool calls, model outputs, citations, caches, telemetry, resources, and user outcomes.
Test	Expected, prohibited, adversarial, malformed, long-context, overloaded, failed, recovery, and rollback cases.
Measure	Task quality, calibration, grounding, safety, latency, throughput, memory, tokens, cost, energy, and operator effort.
Reconcile	Intended system behavior against evaluated model, data, runtime, tools, sources, deployment, and residual risk.

Chapter 02 laboratory and review

Practical laboratory

Write a model-ready change specification and compare results from a vague prompt and the governed specification.

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 executable specifications, formal invariants, and planner-verifier model pairs.

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

Repository context and codebase understanding

Construct high-value context from large, changing repositories.

Chapter operating position

Construct high-value context from large, changing repositories.



3.1 Repository maps and architecture graphs

Repository maps and architecture graphs must be treated as an observable AI-system mechanism rather than a model label, benchmark claim, prompt trick, or framework feature. Index modules, symbols, dependencies, ownership, interfaces, tests, data, and runtime paths. The implementation should name authoritative inputs, data and model versions, identities, state transitions, permissions, resource limits, telemetry, failure behavior, uncertainty, rollback, and acceptance criteria.

This guide traces the subject through the request, specification, repository context, plan, patch, test, review, evidence, release, and user outcome chain. A fluent response, high aggregate score, successful tool call, or green service dashboard is not sufficient proof. Intended behavior must be reconciled with hidden tests, source evidence, negative and adversarial cases, latency and resource measurements, user-visible results, and residual limitations.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one model, dataset, retrieval source, tool, memory tier, optimization, or permission at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated plans and model judgments remain subject to deterministic validation and human authority.

Troubleshooting locates the first divergence from the expected contract or state machine. Account for tokenizer and template mismatch, stale data, context loss, model nondeterminism, caching, quantization, scheduling, permissions, prompt injection, missing telemetry, partial tool effects, and distribution shift. Completion requires reproducibility, causal evidence, validated recovery, and clear disclosure of uncertainty.

Engineering practices

Define the objective, authority, data, model, owner, constraints, risks, and acceptance criteria for repository maps and architecture graphs.

Separate model behavior from retrieval, tools, memory, policy, interface, and human oversight.

Version prompts, schemas, data, model artifacts, runtime configuration, evaluations, and release evidence.

Test nominal, negative, adversarial, long-context, degraded, overloaded, recovery, and rollback conditions.

Retain source, trajectory, tool, citation, metric, profile, decision, and user-outcome evidence.

Prefer least privilege, typed contracts, bounded budgets, staged rollout, abstention, and reversibility.

Failure patterns

Treating repository maps and architecture graphs as a model capability claim disconnected from complete system behavior.

Using one benchmark, model judge, or successful demonstration as proof of production readiness.

Ignoring tokenizer, prompt template, source authority, permissions, cache, hardware, or version boundaries.

Allowing tools, plugins, retrieval content, or memory to redefine trusted instructions.

Optimizing latency, tokens, or cost without measuring quality, safety, and tail failures.

Declaring completion without hidden tests, evidence, residual limitations, rollback, and monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, authority, model, data, prompts, tools, memory, schemas, versions, and assumptions for repository maps and architecture graphs.
Observe	Requests, selected context, trajectories, tool calls, model outputs, citations, caches, telemetry, resources, and user outcomes.
Test	Expected, prohibited, adversarial, malformed, long-context, overloaded, failed, recovery, and rollback cases.
Measure	Task quality, calibration, grounding, safety, latency, throughput, memory, tokens, cost, energy, and operator effort.
Reconcile	Intended system behavior against evaluated model, data, runtime, tools, sources, deployment, and residual risk.

3.2 Semantic and lexical retrieval

Semantic and lexical retrieval must be treated as an observable AI-system mechanism rather than a model label, benchmark claim, prompt trick, or framework feature. Combine file paths, symbols, search, embeddings, call graphs, issue history, and change proximity. The implementation should name authoritative inputs, data and model versions, identities, state transitions, permissions, resource limits, telemetry, failure behavior, uncertainty, rollback, and acceptance criteria.

This guide traces the subject through the request, specification, repository context, plan, patch, test, review, evidence, release, and user outcome chain. A fluent response, high aggregate score, successful tool call, or green service dashboard is not sufficient proof. Intended behavior must be reconciled with hidden tests, source evidence, negative and adversarial cases, latency and resource measurements, user-visible results, and residual limitations.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one model, dataset, retrieval source, tool, memory tier, optimization, or permission at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated plans and model judgments remain subject to deterministic validation and human authority.

Troubleshooting locates the first divergence from the expected contract or state machine. Account for tokenizer and template mismatch, stale data, context loss, model nondeterminism, caching, quantization, scheduling, permissions, prompt injection, missing telemetry, partial tool effects, and distribution shift. Completion requires reproducibility, causal evidence, validated recovery, and clear disclosure of uncertainty.

Engineering practices

Define the objective, authority, data, model, owner, constraints, risks, and acceptance criteria for semantic and lexical retrieval.

Separate model behavior from retrieval, tools, memory, policy, interface, and human oversight.

Version prompts, schemas, data, model artifacts, runtime configuration, evaluations, and release evidence.

Test nominal, negative, adversarial, long-context, degraded, overloaded, recovery, and rollback conditions.

Retain source, trajectory, tool, citation, metric, profile, decision, and user-outcome evidence.

Prefer least privilege, typed contracts, bounded budgets, staged rollout, abstention, and reversibility.

Implementation sketch

```
task:
  objective: add tenant-safe export
  constraints: [no-new-dependency, backward-compatible-api]
  repository_context:
    authoritative: [architecture.md, src/export, tests/export]
    excluded: [secrets, customer-data]
  tracks: [implementation, tests, independent-review]
  completion:
    - hidden-tests-pass
    - cross-tenant-access-denied
    - migration-and-rollback-verified
    - provenance-recorded
```

Failure patterns

Treating semantic and lexical retrieval as a model capability claim disconnected from complete system behavior.

Using one benchmark, model judge, or successful demonstration as proof of production readiness.

Ignoring tokenizer, prompt template, source authority, permissions, cache, hardware, or version boundaries.

Allowing tools, plugins, retrieval content, or memory to redefine trusted instructions.

Optimizing latency, tokens, or cost without measuring quality, safety, and tail failures.

Declaring completion without hidden tests, evidence, residual limitations, rollback, and monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, authority, model, data, prompts, tools, memory, schemas, versions, and assumptions for semantic and lexical retrieval.
Observe	Requests, selected context, trajectories, tool calls, model outputs, citations, caches, telemetry, resources, and user outcomes.
Test	Expected, prohibited, adversarial, malformed, long-context, overloaded, failed, recovery, and rollback cases.
Measure	Task quality, calibration, grounding, safety, latency, throughput, memory, tokens, cost, energy, and operator effort.
Reconcile	Intended system behavior against evaluated model, data, runtime, tools, sources, deployment, and residual risk.



3.3 Context selection and freshness

Context selection and freshness must be treated as an observable AI-system mechanism rather than a model label, benchmark claim, prompt trick, or framework feature. Prefer authoritative, current, dependency-relevant material and expose omissions or stale snapshots. The implementation should name authoritative inputs, data and model versions, identities, state transitions, permissions, resource limits, telemetry, failure behavior, uncertainty, rollback, and acceptance criteria.

This guide traces the subject through the request, specification, repository context, plan, patch, test, review, evidence, release, and user outcome chain. A fluent response, high aggregate score, successful tool call, or green service dashboard is not sufficient proof. Intended behavior must be reconciled with hidden tests, source evidence, negative and adversarial cases, latency and resource measurements, user-visible results, and residual limitations.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one model, dataset, retrieval source, tool, memory tier, optimization, or permission at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated plans and model judgments remain subject to deterministic validation and human authority.

Troubleshooting locates the first divergence from the expected contract or state machine. Account for tokenizer and template mismatch, stale data, context loss, model nondeterminism, caching, quantization, scheduling, permissions, prompt injection, missing telemetry, partial tool effects, and distribution shift. Completion requires reproducibility, causal evidence, validated recovery, and clear disclosure of uncertainty.

Engineering practices

Define the objective, authority, data, model, owner, constraints, risks, and acceptance criteria for context selection and freshness.

Separate model behavior from retrieval, tools, memory, policy, interface, and human oversight.

Version prompts, schemas, data, model artifacts, runtime configuration, evaluations, and release evidence.

Test nominal, negative, adversarial, long-context, degraded, overloaded, recovery, and rollback conditions.

Retain source, trajectory, tool, citation, metric, profile, decision, and user-outcome evidence.

Prefer least privilege, typed contracts, bounded budgets, staged rollout, abstention, and reversibility.

Failure patterns

Treating context selection and freshness as a model capability claim disconnected from complete system behavior.

Using one benchmark, model judge, or successful demonstration as proof of production readiness.

Ignoring tokenizer, prompt template, source authority, permissions, cache, hardware, or version boundaries.

Allowing tools, plugins, retrieval content, or memory to redefine trusted instructions.

Optimizing latency, tokens, or cost without measuring quality, safety, and tail failures.

Declaring completion without hidden tests, evidence, residual limitations, rollback, and monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, authority, model, data, prompts, tools, memory, schemas, versions, and assumptions for context selection and freshness.
Observe	Requests, selected context, trajectories, tool calls, model outputs, citations, caches, telemetry, resources, and user outcomes.
Test	Expected, prohibited, adversarial, malformed, long-context, overloaded, failed, recovery, and rollback cases.
Measure	Task quality, calibration, grounding, safety, latency, throughput, memory, tokens, cost, energy, and operator effort.
Reconcile	Intended system behavior against evaluated model, data, runtime, tools, sources, deployment, and residual risk.

3.4 Sensitive information controls

Sensitive information controls must be treated as an observable AI-system mechanism rather than a model label, benchmark claim, prompt trick, or framework feature. Redact secrets, credentials, customer data, proprietary assets, and restricted directories before model access. The implementation should name authoritative inputs, data and model versions, identities, state transitions, permissions, resource limits, telemetry, failure behavior, uncertainty, rollback, and acceptance criteria.

This guide traces the subject through the request, specification, repository context, plan, patch, test, review, evidence, release, and user outcome chain. A fluent response, high aggregate score, successful tool call, or green service dashboard is not sufficient proof. Intended behavior must be reconciled with hidden tests, source evidence, negative and adversarial cases, latency and resource measurements, user-visible results, and residual limitations.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one model, dataset, retrieval source, tool, memory tier, optimization, or permission at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated plans and model judgments remain subject to deterministic validation and human authority.

Troubleshooting locates the first divergence from the expected contract or state machine. Account for tokenizer and template mismatch, stale data, context loss, model nondeterminism, caching, quantization, scheduling, permissions, prompt injection, missing telemetry, partial tool effects, and distribution shift. Completion requires reproducibility, causal evidence, validated recovery, and clear disclosure of uncertainty.

Engineering practices

Define the objective, authority, data, model, owner, constraints, risks, and acceptance criteria for sensitive information controls.

Separate model behavior from retrieval, tools, memory, policy, interface, and human oversight.

Version prompts, schemas, data, model artifacts, runtime configuration, evaluations, and release evidence.

Test nominal, negative, adversarial, long-context, degraded, overloaded, recovery, and rollback conditions.

Retain source, trajectory, tool, citation, metric, profile, decision, and user-outcome evidence.

Prefer least privilege, typed contracts, bounded budgets, staged rollout, abstention, and reversibility.

Failure patterns

Treating sensitive information controls as a model capability claim disconnected from complete system behavior.

Using one benchmark, model judge, or successful demonstration as proof of production readiness.

Ignoring tokenizer, prompt template, source authority, permissions, cache, hardware, or version boundaries.

Allowing tools, plugins, retrieval content, or memory to redefine trusted instructions.

Optimizing latency, tokens, or cost without measuring quality, safety, and tail failures.

Declaring completion without hidden tests, evidence, residual limitations, rollback, and monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, authority, model, data, prompts, tools, memory, schemas, versions, and assumptions for sensitive information controls.
Observe	Requests, selected context, trajectories, tool calls, model outputs, citations, caches, telemetry, resources, and user outcomes.
Test	Expected, prohibited, adversarial, malformed, long-context, overloaded, failed, recovery, and rollback cases.
Measure	Task quality, calibration, grounding, safety, latency, throughput, memory, tokens, cost, energy, and operator effort.
Reconcile	Intended system behavior against evaluated model, data, runtime, tools, sources, deployment, and residual risk.

Chapter 03 laboratory and review

Practical laboratory

Build a repository context pack for a cross-module change and score relevance, freshness, leakage, and missing 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 compiler-aware context, executable architecture graphs, and continuously synchronized repository twins.

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

Implementation, branching, and parallel agent work

Produce changes safely across isolated workspaces and bounded tasks.

Chapter operating position

Produce changes safely across isolated workspaces and bounded tasks.

4.1 Patch and edit strategies

Patch and edit strategies must be treated as an observable AI-system mechanism rather than a model label, benchmark claim, prompt trick, or framework feature. Choose direct edits, structured transformations, AST changes, code generation, and migration scripts. The implementation should name authoritative inputs, data and model versions, state transitions, permissions, resource limits, telemetry, failure behavior, uncertainty, rollback, and acceptance criteria.

This guide traces the subject through the request, specification, repository context, plan, patch, test, review, evidence, release, and user outcome chain. A fluent response, high aggregate score, successful tool call, or green service dashboard is not sufficient proof. Intended behavior must be reconciled with hidden tests, source evidence, negative and adversarial cases, latency and resource measurements, user-visible results, and residual limitations.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one model, dataset, retrieval source, tool, memory tier, optimization, or permission at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated plans and model judgments remain subject to deterministic validation and human authority.

Troubleshooting locates the first divergence from the expected contract or state machine. Account for tokenizer and template mismatch, stale data, context loss, model nondeterminism, caching, quantization, scheduling, permissions, prompt injection, missing telemetry, partial tool effects, and distribution shift. Completion requires reproducibility, causal evidence, validated recovery, and clear disclosure of uncertainty.

Engineering practices

Define the objective, authority, data, model, owner, constraints, risks, and acceptance criteria for patch and edit strategies.

Separate model behavior from retrieval, tools, memory, policy, interface, and human oversight.

Version prompts, schemas, data, model artifacts, runtime configuration, evaluations, and release evidence.

Test nominal, negative, adversarial, long-context, degraded, overloaded, recovery, and rollback conditions.

Retain source, trajectory, tool, citation, metric, profile, decision, and user-outcome evidence.

Prefer least privilege, typed contracts, bounded budgets, staged rollout, abstention, and reversibility.

Failure patterns

Treating patch and edit strategies as a model capability claim disconnected from complete system behavior.

Using one benchmark, model judge, or successful demonstration as proof of production readiness.

Ignoring tokenizer, prompt template, source authority, permissions, cache, hardware, or version boundaries.

Allowing tools, plugins, retrieval content, or memory to redefine trusted instructions.

Optimizing latency, tokens, or cost without measuring quality, safety, and tail failures.

Declaring completion without hidden tests, evidence, residual limitations, rollback, and monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, authority, model, data, prompts, tools, memory, schemas, versions, and assumptions for patch and edit strategies.
Observe	Requests, selected context, trajectories, tool calls, model outputs, citations, caches, telemetry, resources, and user outcomes.
Test	Expected, prohibited, adversarial, malformed, long-context, overloaded, failed, recovery, and rollback cases.
Measure	Task quality, calibration, grounding, safety, latency, throughput, memory, tokens, cost, energy, and operator effort.
Reconcile	Intended system behavior against evaluated model, data, runtime, tools, sources, deployment, and residual risk.

4.2 Branches, worktrees, and isolation

Branches, worktrees, and isolation must be treated as an observable AI-system mechanism rather than a model label, benchmark claim, prompt trick, or framework feature. Give agents separate workspaces, scoped tasks, bounded credentials, and deterministic integration points. The implementation should name authoritative inputs, data and model versions, identities, state transitions, permissions, resource limits, telemetry, failure behavior, uncertainty, rollback, and acceptance criteria.

This guide traces the subject through the request, specification, repository context, plan, patch, test, review, evidence, release, and user outcome chain. A fluent response, high aggregate score, successful tool call, or green service dashboard is not sufficient proof. Intended behavior must be reconciled with hidden tests, source evidence, negative and adversarial cases, latency and resource measurements, user-visible results, and residual limitations.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one model, dataset, retrieval source, tool, memory tier, optimization, or permission at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated plans and model judgments remain subject to deterministic validation and human authority.

Troubleshooting locates the first divergence from the expected contract or state machine. Account for tokenizer and template mismatch, stale data, context loss, model nondeterminism, caching, quantization, scheduling, permissions, prompt injection, missing telemetry, partial tool effects, and distribution shift. Completion requires reproducibility, causal evidence, validated recovery, and clear disclosure of uncertainty.

Engineering practices

Define the objective, authority, data, model, owner, constraints, risks, and acceptance criteria for branches, worktrees, and isolation.

Separate model behavior from retrieval, tools, memory, policy, interface, and human oversight.

Version prompts, schemas, data, model artifacts, runtime configuration, evaluations, and release evidence.

Test nominal, negative, adversarial, long-context, degraded, overloaded, recovery, and rollback conditions.

Retain source, trajectory, tool, citation, metric, profile, decision, and user-outcome evidence.

Prefer least privilege, typed contracts, bounded budgets, staged rollout, abstention, and reversibility.

Implementation sketch

```
task:
  objective: add tenant-safe export
  constraints: [no-new-dependency, backward-compatible-api]
  repository_context:
    authoritative: [architecture.md, src/export, tests/export]
    excluded: [secrets, customer-data]
  tracks: [implementation, tests, independent-review]
  completion:
    - hidden-tests-pass
    - cross-tenant-access-denied
    - migration-and-rollback-verified
    - provenance-recorded
```

Failure patterns

Treating branches, worktrees, and isolation as a model capability claim disconnected from complete system behavior.

Using one benchmark, model judge, or successful demonstration as proof of production readiness.

Ignoring tokenizer, prompt template, source authority, permissions, cache, hardware, or version boundaries.

Allowing tools, plugins, retrieval content, or memory to redefine trusted instructions.

Optimizing latency, tokens, or cost without measuring quality, safety, and tail failures.

Declaring completion without hidden tests, evidence, residual limitations, rollback, and monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, authority, model, data, prompts, tools, memory, schemas, versions, and assumptions for branches, worktrees, and isolation.
Observe	Requests, selected context, trajectories, tool calls, model outputs, citations, caches, telemetry, resources, and user outcomes.
Test	Expected, prohibited, adversarial, malformed, long-context, overloaded, failed, recovery, and rollback cases.
Measure	Task quality, calibration, grounding, safety, latency, throughput, memory, tokens, cost, energy, and operator effort.
Reconcile	Intended system behavior against evaluated model, data, runtime, tools, sources, deployment, and residual risk.



4.3 Parallel decomposition

Parallel decomposition must be treated as an observable AI-system mechanism rather than a model label, benchmark claim, prompt trick, or framework feature. Run independent research, implementation, tests, documentation, and review tracks with fan-in reconciliation. The implementation should name authoritative inputs, data and model versions, identities, state transitions, permissions, resource limits, telemetry, failure behavior, uncertainty, rollback, and acceptance criteria.

This guide traces the subject through the request, specification, repository context, plan, patch, test, review, evidence, release, and user outcome chain. A fluent response, high aggregate score, successful tool call, or green service dashboard is not sufficient proof. Intended behavior must be reconciled with hidden tests, source evidence, negative and adversarial cases, latency and resource measurements, user-visible results, and residual limitations.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one model, dataset, retrieval source, tool, memory tier, optimization, or permission at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated plans and model judgments remain subject to deterministic validation and human authority.

Troubleshooting locates the first divergence from the expected contract or state machine. Account for tokenizer and template mismatch, stale data, context loss, model nondeterminism, caching, quantization, scheduling, permissions, prompt injection, missing telemetry, partial tool effects, and distribution shift. Completion requires reproducibility, causal evidence, validated recovery, and clear disclosure of uncertainty.

Engineering practices

Define the objective, authority, data, model, owner, constraints, risks, and acceptance criteria for parallel decomposition.

Separate model behavior from retrieval, tools, memory, policy, interface, and human oversight.

Version prompts, schemas, data, model artifacts, runtime configuration, evaluations, and release evidence.

Test nominal, negative, adversarial, long-context, degraded, overloaded, recovery, and rollback conditions.

Retain source, trajectory, tool, citation, metric, profile, decision, and user-outcome evidence.

Prefer least privilege, typed contracts, bounded budgets, staged rollout, abstention, and reversibility.

Failure patterns

Treating parallel decomposition as a model capability claim disconnected from complete system behavior.

Using one benchmark, model judge, or successful demonstration as proof of production readiness.

Ignoring tokenizer, prompt template, source authority, permissions, cache, hardware, or version boundaries.

Allowing tools, plugins, retrieval content, or memory to redefine trusted instructions.

Optimizing latency, tokens, or cost without measuring quality, safety, and tail failures.

Declaring completion without hidden tests, evidence, residual limitations, rollback, and monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, authority, model, data, prompts, tools, memory, schemas, versions, and assumptions for parallel decomposition.
Observe	Requests, selected context, trajectories, tool calls, model outputs, citations, caches, telemetry, resources, and user outcomes.
Test	Expected, prohibited, adversarial, malformed, long-context, overloaded, failed, recovery, and rollback cases.
Measure	Task quality, calibration, grounding, safety, latency, throughput, memory, tokens, cost, energy, and operator effort.
Reconcile	Intended system behavior against evaluated model, data, runtime, tools, sources, deployment, and residual risk.

4.4 Conflict and merge governance

Conflict and merge governance must be treated as an observable AI-system mechanism rather than a model label, benchmark claim, prompt trick, or framework feature. Resolve semantic conflicts, overlapping files, incompatible assumptions, and stale plans before integration. The implementation should name authoritative inputs, data and model versions, identities, state transitions, permissions, resource limits, telemetry, failure behavior, uncertainty, rollback, and acceptance criteria.

This guide traces the subject through the request, specification, repository context, plan, patch, test, review, evidence, release, and user outcome chain. A fluent response, high aggregate score, successful tool call, or green service dashboard is not sufficient proof. Intended behavior must be reconciled with hidden tests, source evidence, negative and adversarial cases, latency and resource measurements, user-visible results, and residual limitations.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one model, dataset, retrieval source, tool, memory tier, optimization, or permission at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated plans and model judgments remain subject to deterministic validation and human authority.

Troubleshooting locates the first divergence from the expected contract or state machine. Account for tokenizer and template mismatch, stale data, context loss, model nondeterminism, caching, quantization, scheduling, permissions, prompt injection, missing telemetry, partial tool effects, and distribution shift. Completion requires reproducibility, causal evidence, validated recovery, and clear disclosure of uncertainty.

Engineering practices

Define the objective, authority, data, model, owner, constraints, risks, and acceptance criteria for conflict and merge governance.

Separate model behavior from retrieval, tools, memory, policy, interface, and human oversight.

Version prompts, schemas, data, model artifacts, runtime configuration, evaluations, and release evidence.

Test nominal, negative, adversarial, long-context, degraded, overloaded, recovery, and rollback conditions.

Retain source, trajectory, tool, citation, metric, profile, decision, and user-outcome evidence.

Prefer least privilege, typed contracts, bounded budgets, staged rollout, abstention, and reversibility.

Failure patterns

Treating conflict and merge governance as a model capability claim disconnected from complete system behavior.

Using one benchmark, model judge, or successful demonstration as proof of production readiness.

Ignoring tokenizer, prompt template, source authority, permissions, cache, hardware, or version boundaries.

Allowing tools, plugins, retrieval content, or memory to redefine trusted instructions.

Optimizing latency, tokens, or cost without measuring quality, safety, and tail failures.

Declaring completion without hidden tests, evidence, residual limitations, rollback, and monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, authority, model, data, prompts, tools, memory, schemas, versions, and assumptions for conflict and merge governance.
Observe	Requests, selected context, trajectories, tool calls, model outputs, citations, caches, telemetry, resources, and user outcomes.
Test	Expected, prohibited, adversarial, malformed, long-context, overloaded, failed, recovery, and rollback cases.
Measure	Task quality, calibration, grounding, safety, latency, throughput, memory, tokens, cost, energy, and operator effort.
Reconcile	Intended system behavior against evaluated model, data, runtime, tools, sources, deployment, and residual risk.

Chapter 04 laboratory and review

Practical laboratory

Run a three-track simulated agent workflow using isolated worktrees and reconcile implementation, tests, and documentation.

Laboratory evidence package

Architecture or topology showing boundaries, identities, dependencies, and authoritative inputs.

Versioned configuration or code with explanatory comments and reproducible setup instructions.

Nominal-path validation plus at least two injected failure or boundary conditions.

Structured logs, traces, metrics, packet captures, test output, or other appropriate evidence.

A concise decision record covering findings, limitations, residual risks, and next actions.

Frontier extension

Explore transactional multi-agent repositories and semantic merge verification.

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

AI-assisted review and security assurance

Use models as reviewers without treating model approval as proof.

Chapter operating position

Use models as reviewers without treating model approval as proof.

5.1 Review dimensions

Review dimensions must be treated as an observable AI-system mechanism rather than a model label, benchmark claim, prompt trick, or framework feature. Inspect correctness, architecture, security, reliability, performance, maintainability, compatibility, and observability. The implementation should name authoritative inputs, data and model versions, identities, state transitions, permissions, resource limits, telemetry, failure behavior, uncertainty, rollback, and acceptance criteria.

This guide traces the subject through the request, specification, repository context, plan, patch, test, review, evidence, release, and user outcome chain. A fluent response, high aggregate score, successful tool call, or green service dashboard is not sufficient proof. Intended behavior must be reconciled with hidden tests, source evidence, negative and adversarial cases, latency and resource measurements, user-visible results, and residual limitations.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one model, dataset, retrieval source, tool, memory tier, optimization, or permission at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated plans and model judgments remain subject to deterministic validation and human authority.

Troubleshooting locates the first divergence from the expected contract or state machine. Account for tokenizer and template mismatch, stale data, context loss, model nondeterminism, caching, quantization, scheduling, permissions, prompt injection, missing telemetry, partial tool effects, and distribution shift. Completion requires reproducibility, causal evidence, validated recovery, and clear disclosure of uncertainty.

Engineering practices

Define the objective, authority, data, model, owner, constraints, risks, and acceptance criteria for review dimensions.

Separate model behavior from retrieval, tools, memory, policy, interface, and human oversight.

Version prompts, schemas, data, model artifacts, runtime configuration, evaluations, and release evidence.

Test nominal, negative, adversarial, long-context, degraded, overloaded, recovery, and rollback conditions.

Retain source, trajectory, tool, citation, metric, profile, decision, and user-outcome evidence.

Prefer least privilege, typed contracts, bounded budgets, staged rollout, abstention, and reversibility.

Failure patterns

Treating review dimensions as a model capability claim disconnected from complete system behavior.

Using one benchmark, model judge, or successful demonstration as proof of production readiness.

Ignoring tokenizer, prompt template, source authority, permissions, cache, hardware, or version boundaries.

Allowing tools, plugins, retrieval content, or memory to redefine trusted instructions.

Optimizing latency, tokens, or cost without measuring quality, safety, and tail failures.

Declaring completion without hidden tests, evidence, residual limitations, rollback, and monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, authority, model, data, prompts, tools, memory, schemas, versions, and assumptions for review dimensions.
Observe	Requests, selected context, trajectories, tool calls, model outputs, citations, caches, telemetry, resources, and user outcomes.
Test	Expected, prohibited, adversarial, malformed, long-context, overloaded, failed, recovery, and rollback cases.
Measure	Task quality, calibration, grounding, safety, latency, throughput, memory, tokens, cost, energy, and operator effort.
Reconcile	Intended system behavior against evaluated model, data, runtime, tools, sources, deployment, and residual risk.

5.2 Adversarial and independent review

Adversarial and independent review must be treated as an observable AI-system mechanism rather than a model label, benchmark claim, prompt trick, or framework feature. Use a model or human that did not author the patch and provide minimized context plus explicit review criteria. The implementation should name authoritative inputs, data and model versions, identities, state transitions, permissions, resource limits, telemetry, failure behavior, uncertainty, rollback, and acceptance criteria.

This guide traces the subject through the request, specification, repository context, plan, patch, test, review, evidence, release, and user outcome chain. A fluent response, high aggregate score, successful tool call, or green service dashboard is not sufficient proof. Intended behavior must be reconciled with hidden tests, source evidence, negative and adversarial cases, latency and resource measurements, user-visible results, and residual limitations.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one model, dataset, retrieval source, tool, memory tier, optimization, or permission at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated plans and model judgments remain subject to deterministic validation and human authority.

Troubleshooting locates the first divergence from the expected contract or state machine. Account for tokenizer and template mismatch, stale data, context loss, model nondeterminism, caching, quantization, scheduling, permissions, prompt injection, missing telemetry, partial tool effects, and distribution shift. Completion requires reproducibility, causal evidence, validated recovery, and clear disclosure of uncertainty.

Engineering practices

Define the objective, authority, data, model, owner, constraints, risks, and acceptance criteria for adversarial and independent review.

Separate model behavior from retrieval, tools, memory, policy, interface, and human oversight.

Version prompts, schemas, data, model artifacts, runtime configuration, evaluations, and release evidence.

Test nominal, negative, adversarial, long-context, degraded, overloaded, recovery, and rollback conditions.

Retain source, trajectory, tool, citation, metric, profile, decision, and user-outcome evidence.

Prefer least privilege, typed contracts, bounded budgets, staged rollout, abstention, and reversibility.

Implementation sketch

```
task:
  objective: add tenant-safe export
  constraints: [no-new-dependency, backward-compatible-api]
  repository_context:
    authoritative: [architecture.md, src/export, tests/export]
    excluded: [secrets, customer-data]
  tracks: [implementation, tests, independent-review]
  completion:
    - hidden-tests-pass
    - cross-tenant-access-denied
    - migration-and-rollback-verified
    - provenance-recorded
```

Failure patterns

Treating adversarial and independent review as a model capability claim disconnected from complete system behavior.

Using one benchmark, model judge, or successful demonstration as proof of production readiness.

Ignoring tokenizer, prompt template, source authority, permissions, cache, hardware, or version boundaries.

Allowing tools, plugins, retrieval content, or memory to redefine trusted instructions.

Optimizing latency, tokens, or cost without measuring quality, safety, and tail failures.

Declaring completion without hidden tests, evidence, residual limitations, rollback, and monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, authority, model, data, prompts, tools, memory, schemas, versions, and assumptions for adversarial and independent review.
Observe	Requests, selected context, trajectories, tool calls, model outputs, citations, caches, telemetry, resources, and user outcomes.
Test	Expected, prohibited, adversarial, malformed, long-context, overloaded, failed, recovery, and rollback cases.
Measure	Task quality, calibration, grounding, safety, latency, throughput, memory, tokens, cost, energy, and operator effort.
Reconcile	Intended system behavior against evaluated model, data, runtime, tools, sources, deployment, and residual risk.



5.3 Prompt injection and repository attacks

Prompt injection and repository attacks must be treated as an observable AI-system mechanism rather than a model label, benchmark claim, prompt trick, or framework feature. Treat source files, issues, comments, generated artifacts, and dependencies as untrusted model inputs. The implementation should name authoritative inputs, data and model versions, identities, state transitions, permissions, resource limits, telemetry, failure behavior, uncertainty, rollback, and acceptance criteria.

This guide traces the subject through the request, specification, repository context, plan, patch, test, review, evidence, release, and user outcome chain. A fluent response, high aggregate score, successful tool call, or green service dashboard is not sufficient proof. Intended behavior must be reconciled with hidden tests, source evidence, negative and adversarial cases, latency and resource measurements, user-visible results, and residual limitations.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one model, dataset, retrieval source, tool, memory tier, optimization, or permission at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated plans and model judgments remain subject to deterministic validation and human authority.

Troubleshooting locates the first divergence from the expected contract or state machine. Account for tokenizer and template mismatch, stale data, context loss, model nondeterminism, caching, quantization, scheduling, permissions, prompt injection, missing telemetry, partial tool effects, and distribution shift. Completion requires reproducibility, causal evidence, validated recovery, and clear disclosure of uncertainty.

Engineering practices

Define the objective, authority, data, model, owner, constraints, risks, and acceptance criteria for prompt injection and repository attacks.

Separate model behavior from retrieval, tools, memory, policy, interface, and human oversight.

Version prompts, schemas, data, model artifacts, runtime configuration, evaluations, and release evidence.

Test nominal, negative, adversarial, long-context, degraded, overloaded, recovery, and rollback conditions.

Retain source, trajectory, tool, citation, metric, profile, decision, and user-outcome evidence.

Prefer least privilege, typed contracts, bounded budgets, staged rollout, abstention, and reversibility.

Failure patterns

Treating prompt injection and repository attacks as a model capability claim disconnected from complete system behavior.

Using one benchmark, model judge, or successful demonstration as proof of production readiness.

Ignoring tokenizer, prompt template, source authority, permissions, cache, hardware, or version boundaries.

Allowing tools, plugins, retrieval content, or memory to redefine trusted instructions.

Optimizing latency, tokens, or cost without measuring quality, safety, and tail failures.

Declaring completion without hidden tests, evidence, residual limitations, rollback, and monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, authority, model, data, prompts, tools, memory, schemas, versions, and assumptions for prompt injection and repository attacks.
Observe	Requests, selected context, trajectories, tool calls, model outputs, citations, caches, telemetry, resources, and user outcomes.
Test	Expected, prohibited, adversarial, malformed, long-context, overloaded, failed, recovery, and rollback cases.
Measure	Task quality, calibration, grounding, safety, latency, throughput, memory, tokens, cost, energy, and operator effort.
Reconcile	Intended system behavior against evaluated model, data, runtime, tools, sources, deployment, and residual risk.

5.4 Review evidence and disposition

Review evidence and disposition must be treated as an observable AI-system mechanism rather than a model label, benchmark claim, prompt trick, or framework feature. Record finding, location, reasoning, severity, evidence, owner, resolution, exception, and retest. The implementation should name authoritative inputs, data and model versions, identities, state transitions, permissions, resource limits, telemetry, failure behavior, uncertainty, rollback, and acceptance criteria.

This guide traces the subject through the request, specification, repository context, plan, patch, test, review, evidence, release, and user outcome chain. A fluent response, high aggregate score, successful tool call, or green service dashboard is not sufficient proof. Intended behavior must be reconciled with hidden tests, source evidence, negative and adversarial cases, latency and resource measurements, user-visible results, and residual limitations.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one model, dataset, retrieval source, tool, memory tier, optimization, or permission at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated plans and model judgments remain subject to deterministic validation and human authority.

Troubleshooting locates the first divergence from the expected contract or state machine. Account for tokenizer and template mismatch, stale data, context loss, model nondeterminism, caching, quantization, scheduling, permissions, prompt injection, missing telemetry, partial tool effects, and distribution shift. Completion requires reproducibility, causal evidence, validated recovery, and clear disclosure of uncertainty.

Engineering practices

Define the objective, authority, data, model, owner, constraints, risks, and acceptance criteria for review evidence and disposition.

Separate model behavior from retrieval, tools, memory, policy, interface, and human oversight.

Version prompts, schemas, data, model artifacts, runtime configuration, evaluations, and release evidence.

Test nominal, negative, adversarial, long-context, degraded, overloaded, recovery, and rollback conditions.

Retain source, trajectory, tool, citation, metric, profile, decision, and user-outcome evidence.

Prefer least privilege, typed contracts, bounded budgets, staged rollout, abstention, and reversibility.

Failure patterns

Treating review evidence and disposition as a model capability claim disconnected from complete system behavior.

Using one benchmark, model judge, or successful demonstration as proof of production readiness.

Ignoring tokenizer, prompt template, source authority, permissions, cache, hardware, or version boundaries.

Allowing tools, plugins, retrieval content, or memory to redefine trusted instructions.

Optimizing latency, tokens, or cost without measuring quality, safety, and tail failures.

Declaring completion without hidden tests, evidence, residual limitations, rollback, and monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, authority, model, data, prompts, tools, memory, schemas, versions, and assumptions for review evidence and disposition.
Observe	Requests, selected context, trajectories, tool calls, model outputs, citations, caches, telemetry, resources, and user outcomes.
Test	Expected, prohibited, adversarial, malformed, long-context, overloaded, failed, recovery, and rollback cases.
Measure	Task quality, calibration, grounding, safety, latency, throughput, memory, tokens, cost, energy, and operator effort.
Reconcile	Intended system behavior against evaluated model, data, runtime, tools, sources, deployment, and residual risk.

Chapter 05 laboratory and review

Practical laboratory

Seed a change with correctness, security, and maintainability defects; compare author-model, reviewer-model, and deterministic-tool findings.

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 proof-producing review agents and adversarial repository content isolation.

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

Generated tests, debugging, and diagnosis

Use models to expand verification and accelerate causal analysis.

Chapter operating position

Use models to expand verification and accelerate causal analysis.



6.1 Test generation

Test generation must be treated as an observable AI-system mechanism rather than a model label, benchmark claim, prompt trick, or framework feature. Create nominal, negative, boundary, property, fuzz, concurrency, migration, and rollback tests from contracts. The implementation should name authoritative inputs, data and model versions, identities, state transitions, permissions, resource limits, telemetry, failure behavior, uncertainty, rollback, and acceptance criteria.

This guide traces the subject through the request, specification, repository context, plan, patch, test, review, evidence, release, and user outcome chain. A fluent response, high aggregate score, successful tool call, or green service dashboard is not sufficient proof. Intended behavior must be reconciled with hidden tests, source evidence, negative and adversarial cases, latency and resource measurements, user-visible results, and residual limitations.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one model, dataset, retrieval source, tool, memory tier, optimization, or permission at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated plans and model judgments remain subject to deterministic validation and human authority.

Troubleshooting locates the first divergence from the expected contract or state machine. Account for tokenizer and template mismatch, stale data, context loss, model nondeterminism, caching, quantization, scheduling, permissions, prompt injection, missing telemetry, partial tool effects, and distribution shift. Completion requires reproducibility, causal evidence, validated recovery, and clear disclosure of uncertainty.

Engineering practices

Define the objective, authority, data, model, owner, constraints, risks, and acceptance criteria for test generation.

Separate model behavior from retrieval, tools, memory, policy, interface, and human oversight.

Version prompts, schemas, data, model artifacts, runtime configuration, evaluations, and release evidence.

Test nominal, negative, adversarial, long-context, degraded, overloaded, recovery, and rollback conditions.

Retain source, trajectory, tool, citation, metric, profile, decision, and user-outcome evidence.

Prefer least privilege, typed contracts, bounded budgets, staged rollout, abstention, and reversibility.

Failure patterns

Treating test generation as a model capability claim disconnected from complete system behavior.

Using one benchmark, model judge, or successful demonstration as proof of production readiness.

Ignoring tokenizer, prompt template, source authority, permissions, cache, hardware, or version boundaries.

Allowing tools, plugins, retrieval content, or memory to redefine trusted instructions.

Optimizing latency, tokens, or cost without measuring quality, safety, and tail failures.

Declaring completion without hidden tests, evidence, residual limitations, rollback, and monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, authority, model, data, prompts, tools, memory, schemas, versions, and assumptions for test generation.
Observe	Requests, selected context, trajectories, tool calls, model outputs, citations, caches, telemetry, resources, and user outcomes.
Test	Expected, prohibited, adversarial, malformed, long-context, overloaded, failed, recovery, and rollback cases.
Measure	Task quality, calibration, grounding, safety, latency, throughput, memory, tokens, cost, energy, and operator effort.
Reconcile	Intended system behavior against evaluated model, data, runtime, tools, sources, deployment, and residual risk.

6.2 Failure reproduction

Failure reproduction must be treated as an observable AI-system mechanism rather than a model label, benchmark claim, prompt trick, or framework feature. Minimize inputs, capture environment, preserve versions, control randomness, and create deterministic fixtures. The implementation should name authoritative inputs, data and model versions, identities, state transitions, permissions, resource limits, telemetry, failure behavior, uncertainty, rollback, and acceptance criteria.

This guide traces the subject through the request, specification, repository context, plan, patch, test, review, evidence, release, and user outcome chain. A fluent response, high aggregate score, successful tool call, or green service dashboard is not sufficient proof. Intended behavior must be reconciled with hidden tests, source evidence, negative and adversarial cases, latency and resource measurements, user-visible results, and residual limitations.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one model, dataset, retrieval source, tool, memory tier, optimization, or permission at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated plans and model judgments remain subject to deterministic validation and human authority.

Troubleshooting locates the first divergence from the expected contract or state machine. Account for tokenizer and template mismatch, stale data, context loss, model nondeterminism, caching, quantization, scheduling, permissions, prompt injection, missing telemetry, partial tool effects, and distribution shift. Completion requires reproducibility, causal evidence, validated recovery, and clear disclosure of uncertainty.

Engineering practices

Define the objective, authority, data, model, owner, constraints, risks, and acceptance criteria for failure reproduction.

Separate model behavior from retrieval, tools, memory, policy, interface, and human oversight.

Version prompts, schemas, data, model artifacts, runtime configuration, evaluations, and release evidence.

Test nominal, negative, adversarial, long-context, degraded, overloaded, recovery, and rollback conditions.

Retain source, trajectory, tool, citation, metric, profile, decision, and user-outcome evidence.

Prefer least privilege, typed contracts, bounded budgets, staged rollout, abstention, and reversibility.

Implementation sketch

```
task:
  objective: add tenant-safe export
  constraints: [no-new-dependency, backward-compatible-api]
  repository_context:
    authoritative: [architecture.md, src/export, tests/export]
    excluded: [secrets, customer-data]
  tracks: [implementation, tests, independent-review]
  completion:
    - hidden-tests-pass
    - cross-tenant-access-denied
    - migration-and-rollback-verified
    - provenance-recorded
```

Failure patterns

Treating failure reproduction as a model capability claim disconnected from complete system behavior.

Using one benchmark, model judge, or successful demonstration as proof of production readiness.

Ignoring tokenizer, prompt template, source authority, permissions, cache, hardware, or version boundaries.

Allowing tools, plugins, retrieval content, or memory to redefine trusted instructions.

Optimizing latency, tokens, or cost without measuring quality, safety, and tail failures.

Declaring completion without hidden tests, evidence, residual limitations, rollback, and monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, authority, model, data, prompts, tools, memory, schemas, versions, and assumptions for failure reproduction.
Observe	Requests, selected context, trajectories, tool calls, model outputs, citations, caches, telemetry, resources, and user outcomes.
Test	Expected, prohibited, adversarial, malformed, long-context, overloaded, failed, recovery, and rollback cases.
Measure	Task quality, calibration, grounding, safety, latency, throughput, memory, tokens, cost, energy, and operator effort.
Reconcile	Intended system behavior against evaluated model, data, runtime, tools, sources, deployment, and residual risk.



6.3 Hypothesis-driven debugging

Hypothesis-driven debugging must be treated as an observable AI-system mechanism rather than a model label, benchmark claim, prompt trick, or framework feature. Generate competing explanations, request evidence, select discriminating tests, and update confidence. The implementation should name authoritative inputs, data and model versions, identities, state transitions, permissions, resource limits, telemetry, failure behavior, uncertainty, rollback, and acceptance criteria.

This guide traces the subject through the request, specification, repository context, plan, patch, test, review, evidence, release, and user outcome chain. A fluent response, high aggregate score, successful tool call, or green service dashboard is not sufficient proof. Intended behavior must be reconciled with hidden tests, source evidence, negative and adversarial cases, latency and resource measurements, user-visible results, and residual limitations.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one model, dataset, retrieval source, tool, memory tier, optimization, or permission at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated plans and model judgments remain subject to deterministic validation and human authority.

Troubleshooting locates the first divergence from the expected contract or state machine. Account for tokenizer and template mismatch, stale data, context loss, model nondeterminism, caching, quantization, scheduling, permissions, prompt injection, missing telemetry, partial tool effects, and distribution shift. Completion requires reproducibility, causal evidence, validated recovery, and clear disclosure of uncertainty.

Engineering practices

Define the objective, authority, data, model, owner, constraints, risks, and acceptance criteria for hypothesis-driven debugging.

Separate model behavior from retrieval, tools, memory, policy, interface, and human oversight.

Version prompts, schemas, data, model artifacts, runtime configuration, evaluations, and release evidence.

Test nominal, negative, adversarial, long-context, degraded, overloaded, recovery, and rollback conditions.

Retain source, trajectory, tool, citation, metric, profile, decision, and user-outcome evidence.

Prefer least privilege, typed contracts, bounded budgets, staged rollout, abstention, and reversibility.

Failure patterns

Treating hypothesis-driven debugging as a model capability claim disconnected from complete system behavior.

Using one benchmark, model judge, or successful demonstration as proof of production readiness.

Ignoring tokenizer, prompt template, source authority, permissions, cache, hardware, or version boundaries.

Allowing tools, plugins, retrieval content, or memory to redefine trusted instructions.

Optimizing latency, tokens, or cost without measuring quality, safety, and tail failures.

Declaring completion without hidden tests, evidence, residual limitations, rollback, and monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, authority, model, data, prompts, tools, memory, schemas, versions, and assumptions for hypothesis-driven debugging.
Observe	Requests, selected context, trajectories, tool calls, model outputs, citations, caches, telemetry, resources, and user outcomes.
Test	Expected, prohibited, adversarial, malformed, long-context, overloaded, failed, recovery, and rollback cases.
Measure	Task quality, calibration, grounding, safety, latency, throughput, memory, tokens, cost, energy, and operator effort.
Reconcile	Intended system behavior against evaluated model, data, runtime, tools, sources, deployment, and residual risk.

6.4 Regression and prevention

Regression and prevention must be treated as an observable AI-system mechanism rather than a model label, benchmark claim, prompt trick, or framework feature. Bind fixes to failing tests, root cause, architecture changes, telemetry, and recurrence monitoring. The implementation should name authoritative inputs, data and model versions, identities, state transitions, permissions, resource limits, telemetry, failure behavior, uncertainty, rollback, and acceptance criteria.

This guide traces the subject through the request, specification, repository context, plan, patch, test, review, evidence, release, and user outcome chain. A fluent response, high aggregate score, successful tool call, or green service dashboard is not sufficient proof. Intended behavior must be reconciled with hidden tests, source evidence, negative and adversarial cases, latency and resource measurements, user-visible results, and residual limitations.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one model, dataset, retrieval source, tool, memory tier, optimization, or permission at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated plans and model judgments remain subject to deterministic validation and human authority.

Troubleshooting locates the first divergence from the expected contract or state machine. Account for tokenizer and template mismatch, stale data, context loss, model nondeterminism, caching, quantization, scheduling, permissions, prompt injection, missing telemetry, partial tool effects, and distribution shift. Completion requires reproducibility, causal evidence, validated recovery, and clear disclosure of uncertainty.

Engineering practices

Define the objective, authority, data, model, owner, constraints, risks, and acceptance criteria for regression and prevention.

Separate model behavior from retrieval, tools, memory, policy, interface, and human oversight.

Version prompts, schemas, data, model artifacts, runtime configuration, evaluations, and release evidence.

Test nominal, negative, adversarial, long-context, degraded, overloaded, recovery, and rollback conditions.

Retain source, trajectory, tool, citation, metric, profile, decision, and user-outcome evidence.

Prefer least privilege, typed contracts, bounded budgets, staged rollout, abstention, and reversibility.

Failure patterns

Treating regression and prevention as a model capability claim disconnected from complete system behavior.

Using one benchmark, model judge, or successful demonstration as proof of production readiness.

Ignoring tokenizer, prompt template, source authority, permissions, cache, hardware, or version boundaries.

Allowing tools, plugins, retrieval content, or memory to redefine trusted instructions.

Optimizing latency, tokens, or cost without measuring quality, safety, and tail failures.

Declaring completion without hidden tests, evidence, residual limitations, rollback, and monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, authority, model, data, prompts, tools, memory, schemas, versions, and assumptions for regression and prevention.
Observe	Requests, selected context, trajectories, tool calls, model outputs, citations, caches, telemetry, resources, and user outcomes.
Test	Expected, prohibited, adversarial, malformed, long-context, overloaded, failed, recovery, and rollback cases.
Measure	Task quality, calibration, grounding, safety, latency, throughput, memory, tokens, cost, energy, and operator effort.
Reconcile	Intended system behavior against evaluated model, data, runtime, tools, sources, deployment, and residual risk.

Chapter 06 laboratory and review

Practical laboratory

Provide an agent with a failing integration scenario and require reproduction, hypotheses, evidence, fix, regression test, and root-cause record.

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-debugging systems with execution traces, verifier loops, and causal program models.

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

CI/CD, provenance, and verified completion

Prevent generated code from bypassing normal delivery controls.

Chapter operating position

Prevent generated code from bypassing normal delivery controls.

7.1 Deterministic quality gates

Deterministic quality gates must be treated as an observable AI-system mechanism rather than a model label, benchmark claim, prompt trick, or framework feature. Run formatting, linting, typing, tests, security analysis, dependency checks, and policy validation. The implementation should name authoritative inputs, data and model versions, identities, state transitions, permissions, resource limits, telemetry, failure behavior, uncertainty, rollback, and acceptance criteria.

This guide traces the subject through the request, specification, repository context, plan, patch, test, review, evidence, release, and user outcome chain. A fluent response, high aggregate score, successful tool call, or green service dashboard is not sufficient proof. Intended behavior must be reconciled with hidden tests, source evidence, negative and adversarial cases, latency and resource measurements, user-visible results, and residual limitations.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one model, dataset, retrieval source, tool, memory tier, optimization, or permission at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated plans and model judgments remain subject to deterministic validation and human authority.

Troubleshooting locates the first divergence from the expected contract or state machine. Account for tokenizer and template mismatch, stale data, context loss, model nondeterminism, caching, quantization, scheduling, permissions, prompt injection, missing telemetry, partial tool effects, and distribution shift. Completion requires reproducibility, causal evidence, validated recovery, and clear disclosure of uncertainty.

Engineering practices

Define the objective, authority, data, model, owner, constraints, risks, and acceptance criteria for deterministic quality gates.

Separate model behavior from retrieval, tools, memory, policy, interface, and human oversight.

Version prompts, schemas, data, model artifacts, runtime configuration, evaluations, and release evidence.

Test nominal, negative, adversarial, long-context, degraded, overloaded, recovery, and rollback conditions.

Retain source, trajectory, tool, citation, metric, profile, decision, and user-outcome evidence.

Prefer least privilege, typed contracts, bounded budgets, staged rollout, abstention, and reversibility.

Failure patterns

Treating deterministic quality gates as a model capability claim disconnected from complete system behavior.

Using one benchmark, model judge, or successful demonstration as proof of production readiness.

Ignoring tokenizer, prompt template, source authority, permissions, cache, hardware, or version boundaries.

Allowing tools, plugins, retrieval content, or memory to redefine trusted instructions.

Optimizing latency, tokens, or cost without measuring quality, safety, and tail failures.

Declaring completion without hidden tests, evidence, residual limitations, rollback, and monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, authority, model, data, prompts, tools, memory, schemas, versions, and assumptions for deterministic quality gates.
Observe	Requests, selected context, trajectories, tool calls, model outputs, citations, caches, telemetry, resources, and user outcomes.
Test	Expected, prohibited, adversarial, malformed, long-context, overloaded, failed, recovery, and rollback cases.
Measure	Task quality, calibration, grounding, safety, latency, throughput, memory, tokens, cost, energy, and operator effort.
Reconcile	Intended system behavior against evaluated model, data, runtime, tools, sources, deployment, and residual risk.

7.2 Build and artifact provenance

Build and artifact provenance must be treated as an observable AI-system mechanism rather than a model label, benchmark claim, prompt trick, or framework feature. Record source revision, model, prompts or task record, tools, dependencies, builder, artifact, and approvals. The implementation should name authoritative inputs, data and model versions, identities, state transitions, permissions, resource limits, telemetry, failure behavior, uncertainty, rollback, and acceptance criteria.

This guide traces the subject through the request, specification, repository context, plan, patch, test, review, evidence, release, and user outcome chain. A fluent response, high aggregate score, successful tool call, or green service dashboard is not sufficient proof. Intended behavior must be reconciled with hidden tests, source evidence, negative and adversarial cases, latency and resource measurements, user-visible results, and residual limitations.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one model, dataset, retrieval source, tool, memory tier, optimization, or permission at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated plans and model judgments remain subject to deterministic validation and human authority.

Troubleshooting locates the first divergence from the expected contract or state machine. Account for tokenizer and template mismatch, stale data, context loss, model nondeterminism, caching, quantization, scheduling, permissions, prompt injection, missing telemetry, partial tool effects, and distribution shift. Completion requires reproducibility, causal evidence, validated recovery, and clear disclosure of uncertainty.

Engineering practices

Define the objective, authority, data, model, owner, constraints, risks, and acceptance criteria for build and artifact provenance.

Separate model behavior from retrieval, tools, memory, policy, interface, and human oversight.

Version prompts, schemas, data, model artifacts, runtime configuration, evaluations, and release evidence.

Test nominal, negative, adversarial, long-context, degraded, overloaded, recovery, and rollback conditions.

Retain source, trajectory, tool, citation, metric, profile, decision, and user-outcome evidence.

Prefer least privilege, typed contracts, bounded budgets, staged rollout, abstention, and reversibility.

Implementation sketch

```
task:
  objective: add tenant-safe export
  constraints: [no-new-dependency, backward-compatible-api]
  repository_context:
    authoritative: [architecture.md, src/export, tests/export]
    excluded: [secrets, customer-data]
  tracks: [implementation, tests, independent-review]
  completion:
    - hidden-tests-pass
    - cross-tenant-access-denied
    - migration-and-rollback-verified
    - provenance-recorded
```

Failure patterns

Treating build and artifact provenance as a model capability claim disconnected from complete system behavior.

Using one benchmark, model judge, or successful demonstration as proof of production readiness.

Ignoring tokenizer, prompt template, source authority, permissions, cache, hardware, or version boundaries.

Allowing tools, plugins, retrieval content, or memory to redefine trusted instructions.

Optimizing latency, tokens, or cost without measuring quality, safety, and tail failures.

Declaring completion without hidden tests, evidence, residual limitations, rollback, and monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, authority, model, data, prompts, tools, memory, schemas, versions, and assumptions for build and artifact provenance.
Observe	Requests, selected context, trajectories, tool calls, model outputs, citations, caches, telemetry, resources, and user outcomes.
Test	Expected, prohibited, adversarial, malformed, long-context, overloaded, failed, recovery, and rollback cases.
Measure	Task quality, calibration, grounding, safety, latency, throughput, memory, tokens, cost, energy, and operator effort.
Reconcile	Intended system behavior against evaluated model, data, runtime, tools, sources, deployment, and residual risk.



7.3 Staged deployment and observation

Staged deployment and observation must be treated as an observable AI-system mechanism rather than a model label, benchmark claim, prompt trick, or framework feature. Use preview, canary, feature flags, migrations, health, SLOs, rollback, and post-deployment tests. The implementation should name authoritative inputs, data and model versions, identities, state transitions, permissions, resource limits, telemetry, failure behavior, uncertainty, rollback, and acceptance criteria.

This guide traces the subject through the request, specification, repository context, plan, patch, test, review, evidence, release, and user outcome chain. A fluent response, high aggregate score, successful tool call, or green service dashboard is not sufficient proof. Intended behavior must be reconciled with hidden tests, source evidence, negative and adversarial cases, latency and resource measurements, user-visible results, and residual limitations.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one model, dataset, retrieval source, tool, memory tier, optimization, or permission at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated plans and model judgments remain subject to deterministic validation and human authority.

Troubleshooting locates the first divergence from the expected contract or state machine. Account for tokenizer and template mismatch, stale data, context loss, model nondeterminism, caching, quantization, scheduling, permissions, prompt injection, missing telemetry, partial tool effects, and distribution shift. Completion requires reproducibility, causal evidence, validated recovery, and clear disclosure of uncertainty.

Engineering practices

Define the objective, authority, data, model, owner, constraints, risks, and acceptance criteria for staged deployment and observation.

Separate model behavior from retrieval, tools, memory, policy, interface, and human oversight.

Version prompts, schemas, data, model artifacts, runtime configuration, evaluations, and release evidence.

Test nominal, negative, adversarial, long-context, degraded, overloaded, recovery, and rollback conditions.

Retain source, trajectory, tool, citation, metric, profile, decision, and user-outcome evidence.

Prefer least privilege, typed contracts, bounded budgets, staged rollout, abstention, and reversibility.

Failure patterns

Treating staged deployment and observation as a model capability claim disconnected from complete system behavior.

Using one benchmark, model judge, or successful demonstration as proof of production readiness.

Ignoring tokenizer, prompt template, source authority, permissions, cache, hardware, or version boundaries.

Allowing tools, plugins, retrieval content, or memory to redefine trusted instructions.

Optimizing latency, tokens, or cost without measuring quality, safety, and tail failures.

Declaring completion without hidden tests, evidence, residual limitations, rollback, and monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, authority, model, data, prompts, tools, memory, schemas, versions, and assumptions for staged deployment and observation.
Observe	Requests, selected context, trajectories, tool calls, model outputs, citations, caches, telemetry, resources, and user outcomes.
Test	Expected, prohibited, adversarial, malformed, long-context, overloaded, failed, recovery, and rollback cases.
Measure	Task quality, calibration, grounding, safety, latency, throughput, memory, tokens, cost, energy, and operator effort.
Reconcile	Intended system behavior against evaluated model, data, runtime, tools, sources, deployment, and residual risk.

7.4 Completion criteria

Completion criteria must be treated as an observable AI-system mechanism rather than a model label, benchmark claim, prompt trick, or framework feature. Require every acceptance criterion, evidence link, unresolved risk, documentation change, and owner decision. The implementation should name authoritative inputs, data and model versions, identities, state transitions, permissions, resource limits, telemetry, failure behavior, uncertainty, rollback, and acceptance criteria.

This guide traces the subject through the request, specification, repository context, plan, patch, test, review, evidence, release, and user outcome chain. A fluent response, high aggregate score, successful tool call, or green service dashboard is not sufficient proof. Intended behavior must be reconciled with hidden tests, source evidence, negative and adversarial cases, latency and resource measurements, user-visible results, and residual limitations.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one model, dataset, retrieval source, tool, memory tier, optimization, or permission at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated plans and model judgments remain subject to deterministic validation and human authority.

Troubleshooting locates the first divergence from the expected contract or state machine. Account for tokenizer and template mismatch, stale data, context loss, model nondeterminism, caching, quantization, scheduling, permissions, prompt injection, missing telemetry, partial tool effects, and distribution shift. Completion requires reproducibility, causal evidence, validated recovery, and clear disclosure of uncertainty.

Engineering practices

Define the objective, authority, data, model, owner, constraints, risks, and acceptance criteria for completion criteria.

Separate model behavior from retrieval, tools, memory, policy, interface, and human oversight.

Version prompts, schemas, data, model artifacts, runtime configuration, evaluations, and release evidence.

Test nominal, negative, adversarial, long-context, degraded, overloaded, recovery, and rollback conditions.

Retain source, trajectory, tool, citation, metric, profile, decision, and user-outcome evidence.

Prefer least privilege, typed contracts, bounded budgets, staged rollout, abstention, and reversibility.

Failure patterns

Treating completion criteria as a model capability claim disconnected from complete system behavior.

Using one benchmark, model judge, or successful demonstration as proof of production readiness.

Ignoring tokenizer, prompt template, source authority, permissions, cache, hardware, or version boundaries.

Allowing tools, plugins, retrieval content, or memory to redefine trusted instructions.

Optimizing latency, tokens, or cost without measuring quality, safety, and tail failures.

Declaring completion without hidden tests, evidence, residual limitations, rollback, and monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, authority, model, data, prompts, tools, memory, schemas, versions, and assumptions for completion criteria.
Observe	Requests, selected context, trajectories, tool calls, model outputs, citations, caches, telemetry, resources, and user outcomes.
Test	Expected, prohibited, adversarial, malformed, long-context, overloaded, failed, recovery, and rollback cases.
Measure	Task quality, calibration, grounding, safety, latency, throughput, memory, tokens, cost, energy, and operator effort.
Reconcile	Intended system behavior against evaluated model, data, runtime, tools, sources, deployment, and residual risk.

Chapter 07 laboratory and review

Practical laboratory

Create a release gate that rejects a generated patch missing provenance, negative tests, migration rollback, or acceptance evidence.

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 signed agent trajectories and machine-verifiable software assurance cases.

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

Economics, governance, and continuous improvement

Operate AI-assisted development as a measured engineering capability.

Chapter operating position

Operate AI-assisted development as a measured engineering capability.

8.1 Token, latency, and cost accounting

Token, latency, and cost accounting must be treated as an observable AI-system mechanism rather than a model label, benchmark claim, prompt trick, or framework feature. Measure model calls, context, retries, cache reuse, human review, compute, and avoided or added work. The implementation should name authoritative inputs, data and model versions, identities, state transitions, permissions, resource limits, telemetry, failure behavior, uncertainty, rollback, and acceptance criteria.

This guide traces the subject through the request, specification, repository context, plan, patch, test, review, evidence, release, and user outcome chain. A fluent response, high aggregate score, successful tool call, or green service dashboard is not sufficient proof. Intended behavior must be reconciled with hidden tests, source evidence, negative and adversarial cases, latency and resource measurements, user-visible results, and residual limitations.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one model, dataset, retrieval source, tool, memory tier, optimization, or permission at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated plans and model judgments remain subject to deterministic validation and human authority.

Troubleshooting locates the first divergence from the expected contract or state machine. Account for tokenizer and template mismatch, stale data, context loss, model nondeterminism, caching, quantization, scheduling, permissions, prompt injection, missing telemetry, partial tool effects, and distribution shift. Completion requires reproducibility, causal evidence, validated recovery, and clear disclosure of uncertainty.

Engineering practices

Define the objective, authority, data, model, owner, constraints, risks, and acceptance criteria for token, latency, and cost accounting.

Separate model behavior from retrieval, tools, memory, policy, interface, and human oversight.

Version prompts, schemas, data, model artifacts, runtime configuration, evaluations, and release evidence.

Test nominal, negative, adversarial, long-context, degraded, overloaded, recovery, and rollback conditions.

Retain source, trajectory, tool, citation, metric, profile, decision, and user-outcome evidence.

Prefer least privilege, typed contracts, bounded budgets, staged rollout, abstention, and reversibility.

Failure patterns

Treating token, latency, and cost accounting as a model capability claim disconnected from complete system behavior.

Using one benchmark, model judge, or successful demonstration as proof of production readiness.

Ignoring tokenizer, prompt template, source authority, permissions, cache, hardware, or version boundaries.

Allowing tools, plugins, retrieval content, or memory to redefine trusted instructions.

Optimizing latency, tokens, or cost without measuring quality, safety, and tail failures.

Declaring completion without hidden tests, evidence, residual limitations, rollback, and monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, authority, model, data, prompts, tools, memory, schemas, versions, and assumptions for token, latency, and cost accounting.
Observe	Requests, selected context, trajectories, tool calls, model outputs, citations, caches, telemetry, resources, and user outcomes.
Test	Expected, prohibited, adversarial, malformed, long-context, overloaded, failed, recovery, and rollback cases.
Measure	Task quality, calibration, grounding, safety, latency, throughput, memory, tokens, cost, energy, and operator effort.
Reconcile	Intended system behavior against evaluated model, data, runtime, tools, sources, deployment, and residual risk.

8.2 Quality and productivity outcomes

Quality and productivity outcomes must be treated as an observable AI-system mechanism rather than a model label, benchmark claim, prompt trick, or framework feature. Measure cycle time, escaped defects, review burden, rework, test coverage quality, incidents, and maintainability. The implementation should name authoritative inputs, data and model versions, identities, state transitions, permissions, resource limits, telemetry, failure behavior, uncertainty, rollback, and acceptance criteria.

This guide traces the subject through the request, specification, repository context, plan, patch, test, review, evidence, release, and user outcome chain. A fluent response, high aggregate score, successful tool call, or green service dashboard is not sufficient proof. Intended behavior must be reconciled with hidden tests, source evidence, negative and adversarial cases, latency and resource measurements, user-visible results, and residual limitations.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one model, dataset, retrieval source, tool, memory tier, optimization, or permission at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated plans and model judgments remain subject to deterministic validation and human authority.

Troubleshooting locates the first divergence from the expected contract or state machine. Account for tokenizer and template mismatch, stale data, context loss, model nondeterminism, caching, quantization, scheduling, permissions, prompt injection, missing telemetry, partial tool effects, and distribution shift. Completion requires reproducibility, causal evidence, validated recovery, and clear disclosure of uncertainty.

Engineering practices

Define the objective, authority, data, model, owner, constraints, risks, and acceptance criteria for quality and productivity outcomes.

Separate model behavior from retrieval, tools, memory, policy, interface, and human oversight.

Version prompts, schemas, data, model artifacts, runtime configuration, evaluations, and release evidence.

Test nominal, negative, adversarial, long-context, degraded, overloaded, recovery, and rollback conditions.

Retain source, trajectory, tool, citation, metric, profile, decision, and user-outcome evidence.

Prefer least privilege, typed contracts, bounded budgets, staged rollout, abstention, and reversibility.

Implementation sketch

```
task:
  objective: add tenant-safe export
  constraints: [no-new-dependency, backward-compatible-api]
  repository_context:
    authoritative: [architecture.md, src/export, tests/export]
    excluded: [secrets, customer-data]
  tracks: [implementation, tests, independent-review]
  completion:
    - hidden-tests-pass
    - cross-tenant-access-denied
    - migration-and-rollback-verified
    - provenance-recorded
```

Failure patterns

Treating quality and productivity outcomes as a model capability claim disconnected from complete system behavior.

Using one benchmark, model judge, or successful demonstration as proof of production readiness.

Ignoring tokenizer, prompt template, source authority, permissions, cache, hardware, or version boundaries.

Allowing tools, plugins, retrieval content, or memory to redefine trusted instructions.

Optimizing latency, tokens, or cost without measuring quality, safety, and tail failures.

Declaring completion without hidden tests, evidence, residual limitations, rollback, and monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, authority, model, data, prompts, tools, memory, schemas, versions, and assumptions for quality and productivity outcomes.
Observe	Requests, selected context, trajectories, tool calls, model outputs, citations, caches, telemetry, resources, and user outcomes.
Test	Expected, prohibited, adversarial, malformed, long-context, overloaded, failed, recovery, and rollback cases.
Measure	Task quality, calibration, grounding, safety, latency, throughput, memory, tokens, cost, energy, and operator effort.
Reconcile	Intended system behavior against evaluated model, data, runtime, tools, sources, deployment, and residual risk.



8.3 Model and workflow evaluation

Model and workflow evaluation must be treated as an observable AI-system mechanism rather than a model label, benchmark claim, prompt trick, or framework feature. Run representative repository tasks, hidden tests, security cases, long-horizon work, and regression suites. The implementation should name authoritative inputs, data and model versions, identities, state transitions, permissions, resource limits, telemetry, failure behavior, uncertainty, rollback, and acceptance criteria.

This guide traces the subject through the request, specification, repository context, plan, patch, test, review, evidence, release, and user outcome chain. A fluent response, high aggregate score, successful tool call, or green service dashboard is not sufficient proof. Intended behavior must be reconciled with hidden tests, source evidence, negative and adversarial cases, latency and resource measurements, user-visible results, and residual limitations.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one model, dataset, retrieval source, tool, memory tier, optimization, or permission at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated plans and model judgments remain subject to deterministic validation and human authority.

Troubleshooting locates the first divergence from the expected contract or state machine. Account for tokenizer and template mismatch, stale data, context loss, model nondeterminism, caching, quantization, scheduling, permissions, prompt injection, missing telemetry, partial tool effects, and distribution shift. Completion requires reproducibility, causal evidence, validated recovery, and clear disclosure of uncertainty.

Engineering practices

Define the objective, authority, data, model, owner, constraints, risks, and acceptance criteria for model and workflow evaluation.

Separate model behavior from retrieval, tools, memory, policy, interface, and human oversight.

Version prompts, schemas, data, model artifacts, runtime configuration, evaluations, and release evidence.

Test nominal, negative, adversarial, long-context, degraded, overloaded, recovery, and rollback conditions.

Retain source, trajectory, tool, citation, metric, profile, decision, and user-outcome evidence.

Prefer least privilege, typed contracts, bounded budgets, staged rollout, abstention, and reversibility.

Failure patterns

Treating model and workflow evaluation as a model capability claim disconnected from complete system behavior.

Using one benchmark, model judge, or successful demonstration as proof of production readiness.

Ignoring tokenizer, prompt template, source authority, permissions, cache, hardware, or version boundaries.

Allowing tools, plugins, retrieval content, or memory to redefine trusted instructions.

Optimizing latency, tokens, or cost without measuring quality, safety, and tail failures.

Declaring completion without hidden tests, evidence, residual limitations, rollback, and monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, authority, model, data, prompts, tools, memory, schemas, versions, and assumptions for model and workflow evaluation.
Observe	Requests, selected context, trajectories, tool calls, model outputs, citations, caches, telemetry, resources, and user outcomes.
Test	Expected, prohibited, adversarial, malformed, long-context, overloaded, failed, recovery, and rollback cases.
Measure	Task quality, calibration, grounding, safety, latency, throughput, memory, tokens, cost, energy, and operator effort.
Reconcile	Intended system behavior against evaluated model, data, runtime, tools, sources, deployment, and residual risk.

8.4 Policy and learning loop

Policy and learning loop must be treated as an observable AI-system mechanism rather than a model label, benchmark claim, prompt trick, or framework feature. Update approved models, tools, prompts, context rules, permissions, and training material from evidence. The implementation should name authoritative inputs, data and model versions, identities, state transitions, permissions, resource limits, telemetry, failure behavior, uncertainty, rollback, and acceptance criteria.

This guide traces the subject through the request, specification, repository context, plan, patch, test, review, evidence, release, and user outcome chain. A fluent response, high aggregate score, successful tool call, or green service dashboard is not sufficient proof. Intended behavior must be reconciled with hidden tests, source evidence, negative and adversarial cases, latency and resource measurements, user-visible results, and residual limitations.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one model, dataset, retrieval source, tool, memory tier, optimization, or permission at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated plans and model judgments remain subject to deterministic validation and human authority.

Troubleshooting locates the first divergence from the expected contract or state machine. Account for tokenizer and template mismatch, stale data, context loss, model nondeterminism, caching, quantization, scheduling, permissions, prompt injection, missing telemetry, partial tool effects, and distribution shift. Completion requires reproducibility, causal evidence, validated recovery, and clear disclosure of uncertainty.

Engineering practices

Define the objective, authority, data, model, owner, constraints, risks, and acceptance criteria for policy and learning loop.

Separate model behavior from retrieval, tools, memory, policy, interface, and human oversight.

Version prompts, schemas, data, model artifacts, runtime configuration, evaluations, and release evidence.

Test nominal, negative, adversarial, long-context, degraded, overloaded, recovery, and rollback conditions.

Retain source, trajectory, tool, citation, metric, profile, decision, and user-outcome evidence.

Prefer least privilege, typed contracts, bounded budgets, staged rollout, abstention, and reversibility.

Failure patterns

Treating policy and learning loop as a model capability claim disconnected from complete system behavior.

Using one benchmark, model judge, or successful demonstration as proof of production readiness.

Ignoring tokenizer, prompt template, source authority, permissions, cache, hardware, or version boundaries.

Allowing tools, plugins, retrieval content, or memory to redefine trusted instructions.

- Optimizing latency, tokens, or cost without measuring quality, safety, and tail failures.
- Declaring completion without hidden tests, evidence, residual limitations, rollback, and monitoring.

Validation and evidence

Method	Expected evidence
Examine	Requirements, authority, model, data, prompts, tools, memory, schemas, versions, and assumptions for policy and learning loop.
Observe	Requests, selected context, trajectories, tool calls, model outputs, citations, caches, telemetry, resources, and user outcomes.
Test	Expected, prohibited, adversarial, malformed, long-context, overloaded, failed, recovery, and rollback cases.
Measure	Task quality, calibration, grounding, safety, latency, throughput, memory, tokens, cost, energy, and operator effort.
Reconcile	Intended system behavior against evaluated model, data, runtime, tools, sources, deployment, and residual risk.

Chapter 08 laboratory and review

Practical laboratory

Benchmark two AI-assisted workflows on the same repository tasks and compare verified outcomes rather than raw completion speed.

Laboratory evidence package

Architecture or topology showing boundaries, identities, dependencies, and authoritative inputs.

Versioned configuration or code with explanatory comments and reproducible setup instructions.

Nominal-path validation plus at least two injected failure or boundary conditions.

Structured logs, traces, metrics, packet captures, test output, or other appropriate evidence.

A concise decision record covering findings, limitations, residual risks, and next actions.

Frontier extension

Explore adaptive model routing, organization-specific coding evaluators, and self-improving governed workflows.

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. NIST - SP 800-218A - Secure Software Development Practices for Generative AI and Dual-Use Foundation Models

Role: AI-specific secure development profile.

Verified: 2026-07-26

Official source: <https://csrc.nist.gov/pubs/sp/800/218/a/final>

02. 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>

03. OWASP GenAI Security Project - OWASP Top 10 for LLM and GenAI Applications 2025

Role: Risk and mitigation baseline for generative-AI applications.

Verified: 2026-07-26

Official source: <https://genai.owasp.org/llm-top-10/>

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

Role: Agentic-AI security risks and mitigations.

Verified: 2026-07-26

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

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

Role: Authoritative protocol requirements for resources, prompts, tools, transports, authorization, and lifecycle.

Verified: 2026-07-26

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

06. OpenAPI Initiative - OpenAPI Specification

Role: Machine-readable HTTP API contracts.

Verified: 2026-07-26

Official source: <https://spec.openapis.org/oas/latest.html>

07. JSON Schema - JSON Schema 2020-12

Role: Typed JSON instance and schema validation.

Verified: 2026-07-26

Official source: <https://json-schema.org/draft/2020-12>

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-SWE - Software Engineering & Design
Secondary domain	MYTHOS-AI; MYTHOS-SEC; MYTHOS-DAT
Publication class	Field Guide / Canonical Living Overview Guide
Skill levels	L1 Foundational; L2 Practitioner; L3 Advanced; L4 Frontier
Tags	AI-assisted development, coding agents, repository context, code review, test generation, multi-agent development, provenance, CI/CD, prompt injection, verified completion
Canonical route	/library/field-guides/mythos-fg-swe-0020/

Living publication

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