CANONICAL LIVING OVERVIEW GUIDE

Error Architecture, Reliability, and Failure Semantics

A reliability guide covering error taxonomies, typed failures, exceptions and results, fault containment, timeouts, retries, backpressure, overload, circuit breakers, idempotency, partial failure, degradation, recovery, observability, error budgets, resilience testing, and failure-safe API and workflow semantics.

PERMANENT ID
MYTHOS-FG-SWE-0004

PUBLICATION CLASS
FIELD GUIDE

VERSION / STATUS
1.0.0-candidate / CANDIDATE

PERMANENT PUBLICATION IDENTITY

Error Architecture, Reliability, and Failure Semantics

Document ID
MYTHOS-FG-SWE-0004

Version
1.0.0-candidate

Status
Candidate Focused Field Guide

Review date
2026-10-26

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

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

Reader orientation

Dimension	Engineering position
Primary domain	MYTHOS-SWE - Software Engineering & Design
Secondary domain	MYTHOS-CLD; MYTHOS-DAT; MYTHOS-SEC
Audience	Software, platform, distributed-systems, SRE, API, data, cloud, and security engineers.
Prerequisites	Programming, APIs, concurrency, distributed systems, observability, and software architecture fundamentals.
Publisher / owner	Mythos Systems / Aaron Stovall
Public classification	Public technical guidance

Expected outcomes

Design errors as part of contracts and state machines rather than incidental strings or stack traces.

Differentiate faults, errors, failures, defects, invalid input, conflicts, timeouts, overload, and unknown outcomes.

Contain failure through boundaries, budgets, deadlines, admission control, isolation, and graceful degradation.

Use retries, circuit breaking, idempotency, compensation, and recovery only when their semantics are proven safe.

Measure reliability through user outcomes, SLOs, error budgets, failure tests, and causal evidence.

How to use this guide

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

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

Learning and assurance model

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

Contents

CHAPTER 01

Failure model and error taxonomy

Create a shared language for abnormal conditions and their operational consequences.

Chapter operating position
Create a shared language for abnormal conditions and their operational consequences.

1.1 Faults, errors, failures, and defects

Faults, errors, failures, and defects must be treated as an observable engineering mechanism rather than a slogan, framework checkbox, or tool output. Separate underlying cause, corrupted state, externally visible service failure, and implementation or design defect. The design should identify authoritative inputs, states, boundaries, assumptions, dependencies, limits, ownership, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the fault, detected condition, typed error, propagation, containment, recovery action, telemetry, and user outcome chain. A diagram, passing test, successful request, exploit result, benchmark, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with implemented state, negative and boundary tests, degraded conditions, raw evidence, recovery, and the resulting user, service, or risk outcome.

Implementation begins with a minimal known-good baseline and explicit invariants. Introduce one variable or dependency at a time, preserve before-state evidence, test expected and prohibited behavior, inject realistic failure, and retain a reversible path. Automation and AI assistance must stop safely when authority, freshness, identity, state, or expected outcomes are ambiguous.

Troubleshooting and assurance should locate the first divergence from the expected contract or state machine. Account for time, concurrency, caching, eventual consistency, data and schema drift, partial failure, environment differences, hidden coupling, resource saturation, and missing telemetry. Completion requires reproducible evidence, a causal explanation, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

- Define the objective, owner, authority, scope, assumptions, dependencies, and acceptance criteria for faults, errors, failures, and defects.
- Model expected states, boundaries, inputs, outputs, failure modes, and evidence before implementation.
- Validate intended design, implemented behavior, runtime state, and user or mission outcome independently.
- Test positive, negative, boundary, degraded, failed, recovery, scale, and rollback conditions.
- Preserve source, version, configuration, data, telemetry, artifacts, timestamps, decisions, and limitations.
- Automate through typed contracts, least privilege, deterministic gates, bounded concurrency, canaries, and reversibility.

Failure patterns

- Treating faults, errors, failures, and defects as a document or tool activity disconnected from actual system behavior.
- Relying on intended design or configuration without proving implementation and runtime outcomes.
- Ignoring failure, concurrency, data, identity, environment, compatibility, and operational boundaries.
- Changing several variables before preserving the original state and stating a falsifiable hypothesis.
- Allowing generated code, tests, findings, or optimizations to proceed without authoritative inputs and review.
- Declaring success after one positive example without negative tests, reproducibility, rollback, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, contracts, data, versions, ownership, and assumptions for faults, errors, failures, and defects.
Observe	Runtime state, control flow, telemetry, artifacts, resource behavior, errors, test results, and user-visible outcomes.
Test	Expected, prohibited, boundary, malformed, concurrent, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, resource use, reliability, coverage, risk, maintainability, and operational impact.
Reconcile	Intended requirements and design against code, configuration, runtime behavior, tests, evidence, and residual limitations.

1.2 Client, domain, dependency, and platform errors

Client, domain, dependency, and platform errors must be treated as an observable engineering mechanism rather than a slogan, framework checkbox, or tool output. Classify invalid requests, business conflicts, authorization, resource limits, external dependencies, infrastructure, and bugs. The design should identify authoritative inputs, states, boundaries, assumptions, dependencies, limits, ownership, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the fault, detected condition, typed error, propagation, containment, recovery action, telemetry, and user outcome chain. A diagram, passing test, successful request, exploit result, benchmark, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with implemented state, negative and boundary tests, degraded conditions, raw evidence, recovery, and the resulting user, service, or risk outcome.

Implementation begins with a minimal known-good baseline and explicit invariants. Introduce one variable or dependency at a time, preserve before-state evidence, test expected and prohibited behavior, inject realistic failure, and retain a reversible path. Automation and AI assistance must stop safely when authority, freshness, identity, state, or expected outcomes are ambiguous.

Troubleshooting and assurance should locate the first divergence from the expected contract or state machine. Account for time, concurrency, caching, eventual consistency, data and schema drift, partial failure, environment differences, hidden coupling, resource saturation, and missing telemetry. Completion requires reproducible evidence, a causal explanation, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

- Define the objective, owner, authority, scope, assumptions, dependencies, and acceptance criteria for client, domain, dependency, and platform errors.
- Model expected states, boundaries, inputs, outputs, failure modes, and evidence before implementation.
- Validate intended design, implemented behavior, runtime state, and user or mission outcome independently.
- Test positive, negative, boundary, degraded, failed, recovery, scale, and rollback conditions.
- Preserve source, version, configuration, data, telemetry, artifacts, timestamps, decisions, and limitations.
- Automate through typed contracts, least privilege, deterministic gates, bounded concurrency, canaries, and reversibility.

Implementation sketch

```
error_contract:
  type: dependency-timeout
  retryable: true
  operation_outcome: unknown
  retry_policy:
    max_attempts: 2
    budget_from_parent_deadline: true
    jitter: full
  idempotency_key: required
  telemetry: [dependency, elapsed, deadline, attempt, effect_state]
  fallback: explicit-degraded-response
```

Failure patterns

- Treating client, domain, dependency, and platform errors as a document or tool activity disconnected from actual system behavior.
- Relying on intended design or configuration without proving implementation and runtime outcomes.

Ignoring failure, concurrency, data, identity, environment, compatibility, and operational boundaries.

Changing several variables before preserving the original state and stating a falsifiable hypothesis.

Allowing generated code, tests, findings, or optimizations to proceed without authoritative inputs and review.

Declaring success after one positive example without negative tests, reproducibility, rollback, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, contracts, data, versions, ownership, and assumptions for client, domain, dependency, and platform errors.
Observe	Runtime state, control flow, telemetry, artifacts, resource behavior, errors, test results, and user-visible outcomes.
Test	Expected, prohibited, boundary, malformed, concurrent, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, resource use, reliability, coverage, risk, maintainability, and operational impact.
Reconcile	Intended requirements and design against code, configuration, runtime behavior, tests, evidence, and residual limitations.

1.3 Transient, permanent, and unknown outcomes

Transient, permanent, and unknown outcomes must be treated as an observable engineering mechanism rather than a slogan, framework checkbox, or tool output. Identify retryable conditions, deterministic rejection, ambiguous completion, partial effects, and recovery needs. The design should identify authoritative inputs, states, boundaries, assumptions, dependencies, limits, ownership, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the fault, detected condition, typed error, propagation, containment, recovery action, telemetry, and user outcome chain. A diagram, passing test, successful request, exploit result, benchmark, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with implemented state, negative and boundary tests, degraded conditions, raw evidence, recovery, and the resulting user, service, or risk outcome.

Implementation begins with a minimal known-good baseline and explicit invariants. Introduce one variable or dependency at a time, preserve before-state evidence, test expected and prohibited behavior, inject realistic failure, and retain a reversible path. Automation and AI assistance must stop safely when authority, freshness, identity, state, or expected outcomes are ambiguous.

Troubleshooting and assurance should locate the first divergence from the expected contract or state machine. Account for time, concurrency, caching, eventual consistency, data and schema drift, partial failure, environment differences, hidden coupling, resource saturation, and missing telemetry. Completion requires reproducible evidence, a causal explanation, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

Define the objective, owner, authority, scope, assumptions, dependencies, and acceptance criteria for transient, permanent, and unknown outcomes.

Model expected states, boundaries, inputs, outputs, failure modes, and evidence before implementation.

Validate intended design, implemented behavior, runtime state, and user or mission outcome independently.

Test positive, negative, boundary, degraded, failed, recovery, scale, and rollback conditions.

Preserve source, version, configuration, data, telemetry, artifacts, timestamps, decisions, and limitations.

Automate through typed contracts, least privilege, deterministic gates, bounded concurrency, canaries, and reversibility.

Failure patterns

Treating transient, permanent, and unknown outcomes as a document or tool activity disconnected from actual system behavior.

Relying on intended design or configuration without proving implementation and runtime outcomes.

Ignoring failure, concurrency, data, identity, environment, compatibility, and operational boundaries.

Changing several variables before preserving the original state and stating a falsifiable hypothesis.

Allowing generated code, tests, findings, or optimizations to proceed without authoritative inputs and review.

Declaring success after one positive example without negative tests, reproducibility, rollback, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, contracts, data, versions, ownership, and assumptions for transient, permanent, and unknown outcomes.
Observe	Runtime state, control flow, telemetry, artifacts, resource behavior, errors, test results, and user-visible outcomes.
Test	Expected, prohibited, boundary, malformed, concurrent, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, resource use, reliability, coverage, risk, maintainability, and operational impact.
Reconcile	Intended requirements and design against code, configuration, runtime behavior, tests, evidence, and residual limitations.

1.4 Severity, ownership, and exposure

Severity, ownership, and exposure must be treated as an observable engineering mechanism rather than a slogan, framework checkbox, or tool output. Define user impact, blast radius, data risk, security significance, responsible component, and safe information disclosure. The design should identify authoritative inputs, states, boundaries, assumptions, dependencies, limits, ownership, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the fault, detected condition, typed error, propagation, containment, recovery action, telemetry, and user outcome chain. A diagram, passing test, successful request, exploit result, benchmark, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with implemented state, negative and boundary tests, degraded conditions, raw evidence, recovery, and the resulting user, service, or risk outcome.

Implementation begins with a minimal known-good baseline and explicit invariants. Introduce one variable or dependency at a time, preserve before-state evidence, test expected and prohibited behavior, inject realistic failure, and retain a reversible path. Automation and AI assistance must stop safely when authority, freshness, identity, state, or expected outcomes are ambiguous.

Troubleshooting and assurance should locate the first divergence from the expected contract or state machine. Account for time, concurrency, caching, eventual consistency, data and schema drift, partial failure, environment differences, hidden coupling, resource saturation, and missing telemetry. Completion requires reproducible evidence, a causal explanation, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

Define the objective, owner, authority, scope, assumptions, dependencies, and acceptance criteria for severity, ownership, and exposure.

Model expected states, boundaries, inputs, outputs, failure modes, and evidence before implementation.

Validate intended design, implemented behavior, runtime state, and user or mission outcome independently.

Test positive, negative, boundary, degraded, failed, recovery, scale, and rollback conditions.

Preserve source, version, configuration, data, telemetry, artifacts, timestamps, decisions, and limitations.

Automate through typed contracts, least privilege, deterministic gates, bounded concurrency, canaries, and reversibility.

Failure patterns

Treating severity, ownership, and exposure as a document or tool activity disconnected from actual system behavior.

Relying on intended design or configuration without proving implementation and runtime outcomes.

Ignoring failure, concurrency, data, identity, environment, compatibility, and operational boundaries.

Changing several variables before preserving the original state and stating a falsifiable hypothesis.

Allowing generated code, tests, findings, or optimizations to proceed without authoritative inputs and review.

Declaring success after one positive example without negative tests, reproducibility, rollback, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, contracts, data, versions, ownership, and assumptions for severity, ownership, and exposure.
Observe	Runtime state, control flow, telemetry, artifacts, resource behavior, errors, test results, and user-visible outcomes.
Test	Expected, prohibited, boundary, malformed, concurrent, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, resource use, reliability, coverage, risk, maintainability, and operational impact.
Reconcile	Intended requirements and design against code, configuration, runtime behavior, tests, evidence, and residual limitations.

Chapter 01 laboratory and review

Practical laboratory

Create an error taxonomy for a distributed application and classify twenty scenarios, including ambiguous and partial outcomes.

Laboratory evidence package

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

Frontier extension

Extend failure models to agent plans, model uncertainty, cyber-physical safety, confidential workloads, and quantum-era dependencies.

Review gate

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

CHAPTER 02

Error representation and program structure

Represent failures so callers can reason, recover, and preserve causal information.

Chapter operating position
Represent failures so callers can reason, recover, and preserve causal information.

2.1 Exceptions, result types, and error values

Exceptions, result types, and error values must be treated as an observable engineering mechanism rather than a slogan, framework checkbox, or tool output. Choose typed alternatives, checked propagation, exceptions, sentinel values, status objects, and language-specific idioms. The design should identify authoritative inputs, states, boundaries, assumptions, dependencies, limits, ownership, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the fault, detected condition, typed error, propagation, containment, recovery action, telemetry, and user outcome chain. A diagram, passing test, successful request, exploit result, benchmark, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with implemented state, negative and boundary tests, degraded conditions, raw evidence, recovery, and the resulting user, service, or risk outcome.

Implementation begins with a minimal known-good baseline and explicit invariants. Introduce one variable or dependency at a time, preserve before-state evidence, test expected and prohibited behavior, inject realistic failure, and retain a reversible path. Automation and AI assistance must stop safely when authority, freshness, identity, state, or expected outcomes are ambiguous.

Troubleshooting and assurance should locate the first divergence from the expected contract or state machine. Account for time, concurrency, caching, eventual consistency, data and schema drift, partial failure, environment differences, hidden coupling, resource saturation, and missing telemetry. Completion requires reproducible evidence, a causal explanation, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

- Define the objective, owner, authority, scope, assumptions, dependencies, and acceptance criteria for exceptions, result types, and error values.
- Model expected states, boundaries, inputs, outputs, failure modes, and evidence before implementation.
- Validate intended design, implemented behavior, runtime state, and user or mission outcome independently.
- Test positive, negative, boundary, degraded, failed, recovery, scale, and rollback conditions.
- Preserve source, version, configuration, data, telemetry, artifacts, timestamps, decisions, and limitations.
- Automate through typed contracts, least privilege, deterministic gates, bounded concurrency, canaries, and reversibility.

Failure patterns

- Treating exceptions, result types, and error values as a document or tool activity disconnected from actual system behavior.
- Relying on intended design or configuration without proving implementation and runtime outcomes.
- Ignoring failure, concurrency, data, identity, environment, compatibility, and operational boundaries.
- Changing several variables before preserving the original state and stating a falsifiable hypothesis.
- Allowing generated code, tests, findings, or optimizations to proceed without authoritative inputs and review.
- Declaring success after one positive example without negative tests, reproducibility, rollback, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, contracts, data, versions, ownership, and assumptions for exceptions, result types, and error values.
Observe	Runtime state, control flow, telemetry, artifacts, resource behavior, errors, test results, and user-visible outcomes.
Test	Expected, prohibited, boundary, malformed, concurrent, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, resource use, reliability, coverage, risk, maintainability, and operational impact.
Reconcile	Intended requirements and design against code, configuration, runtime behavior, tests, evidence, and residual limitations.

2.2 Context, cause, and error chains

Context, cause, and error chains must be treated as an observable engineering mechanism rather than a slogan, framework checkbox, or tool output. Preserve operation, identity, resource, dependency, original cause, stack, source, and correlation without duplication. The design should identify authoritative inputs, states, boundaries, assumptions, dependencies, limits, ownership, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the fault, detected condition, typed error, propagation, containment, recovery action, telemetry, and user outcome chain. A diagram, passing test, successful request, exploit result, benchmark, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with implemented state, negative and boundary tests, degraded conditions, raw evidence, recovery, and the resulting user, service, or risk outcome.

Implementation begins with a minimal known-good baseline and explicit invariants. Introduce one variable or dependency at a time, preserve before-state evidence, test expected and prohibited behavior, inject realistic failure, and retain a reversible path. Automation and AI assistance must stop safely when authority, freshness, identity, state, or expected outcomes are ambiguous.

Troubleshooting and assurance should locate the first divergence from the expected contract or state machine. Account for time, concurrency, caching, eventual consistency, data and schema drift, partial failure, environment differences, hidden coupling, resource saturation, and missing telemetry. Completion requires reproducible evidence, a causal explanation, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

- Define the objective, owner, authority, scope, assumptions, dependencies, and acceptance criteria for context, cause, and error chains.
- Model expected states, boundaries, inputs, outputs, failure modes, and evidence before implementation.
- Validate intended design, implemented behavior, runtime state, and user or mission outcome independently.
- Test positive, negative, boundary, degraded, failed, recovery, scale, and rollback conditions.
- Preserve source, version, configuration, data, telemetry, artifacts, timestamps, decisions, and limitations.
- Automate through typed contracts, least privilege, deterministic gates, bounded concurrency, canaries, and reversibility.

Implementation sketch

```
error_contract:
  type: dependency-timeout
  retryable: true
  operation_outcome: unknown
  retry_policy:
    max_attempts: 2
    budget_from_parent_deadline: true
    jitter: full
  idempotency_key: required
  telemetry: [dependency, elapsed, deadline, attempt, effect_state]
  fallback: explicit-degraded-response
```

Failure patterns

- Treating context, cause, and error chains as a document or tool activity disconnected from actual system behavior.
- Relying on intended design or configuration without proving implementation and runtime outcomes.

Ignoring failure, concurrency, data, identity, environment, compatibility, and operational boundaries.

Changing several variables before preserving the original state and stating a falsifiable hypothesis.

Allowing generated code, tests, findings, or optimizations to proceed without authoritative inputs and review.

Declaring success after one positive example without negative tests, reproducibility, rollback, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, contracts, data, versions, ownership, and assumptions for context, cause, and error chains.
Observe	Runtime state, control flow, telemetry, artifacts, resource behavior, errors, test results, and user-visible outcomes.
Test	Expected, prohibited, boundary, malformed, concurrent, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, resource use, reliability, coverage, risk, maintainability, and operational impact.
Reconcile	Intended requirements and design against code, configuration, runtime behavior, tests, evidence, and residual limitations.

2.3 Boundary translation

Boundary translation must be treated as an observable engineering mechanism rather than a slogan, framework checkbox, or tool output. Map low-level failures into domain, API, message, CLI, UI, and operational meanings while retaining diagnostics. The design should identify authoritative inputs, states, boundaries, assumptions, dependencies, limits, ownership, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the fault, detected condition, typed error, propagation, containment, recovery action, telemetry, and user outcome chain. A diagram, passing test, successful request, exploit result, benchmark, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with implemented state, negative and boundary tests, degraded conditions, raw evidence, recovery, and the resulting user, service, or risk outcome.

Implementation begins with a minimal known-good baseline and explicit invariants. Introduce one variable or dependency at a time, preserve before-state evidence, test expected and prohibited behavior, inject realistic failure, and retain a reversible path. Automation and AI assistance must stop safely when authority, freshness, identity, state, or expected outcomes are ambiguous.

Troubleshooting and assurance should locate the first divergence from the expected contract or state machine. Account for time, concurrency, caching, eventual consistency, data and schema drift, partial failure, environment differences, hidden coupling, resource saturation, and missing telemetry. Completion requires reproducible evidence, a causal explanation, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

Define the objective, owner, authority, scope, assumptions, dependencies, and acceptance criteria for boundary translation.

Model expected states, boundaries, inputs, outputs, failure modes, and evidence before implementation.

Validate intended design, implemented behavior, runtime state, and user or mission outcome independently.

Test positive, negative, boundary, degraded, failed, recovery, scale, and rollback conditions.

Preserve source, version, configuration, data, telemetry, artifacts, timestamps, decisions, and limitations.

Automate through typed contracts, least privilege, deterministic gates, bounded concurrency, canaries, and reversibility.

Failure patterns

Treating boundary translation as a document or tool activity disconnected from actual system behavior.

Relying on intended design or configuration without proving implementation and runtime outcomes.

Ignoring failure, concurrency, data, identity, environment, compatibility, and operational boundaries.

Changing several variables before preserving the original state and stating a falsifiable hypothesis.

Allowing generated code, tests, findings, or optimizations to proceed without authoritative inputs and review.

Declaring success after one positive example without negative tests, reproducibility, rollback, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, contracts, data, versions, ownership, and assumptions for boundary translation.
Observe	Runtime state, control flow, telemetry, artifacts, resource behavior, errors, test results, and user-visible outcomes.
Test	Expected, prohibited, boundary, malformed, concurrent, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, resource use, reliability, coverage, risk, maintainability, and operational impact.
Reconcile	Intended requirements and design against code, configuration, runtime behavior, tests, evidence, and residual limitations.

2.4 Invariants and impossible states

Invariants and impossible states must be treated as an observable engineering mechanism rather than a slogan, framework checkbox, or tool output. Use types, validation, construction, ownership, state machines, and assertions to prevent or reveal invalid state. The design should identify authoritative inputs, states, boundaries, assumptions, dependencies, limits, ownership, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the fault, detected condition, typed error, propagation, containment, recovery action, telemetry, and user outcome chain. A diagram, passing test, successful request, exploit result, benchmark, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with implemented state, negative and boundary tests, degraded conditions, raw evidence, recovery, and the resulting user, service, or risk outcome.

Implementation begins with a minimal known-good baseline and explicit invariants. Introduce one variable or dependency at a time, preserve before-state evidence, test expected and prohibited behavior, inject realistic failure, and retain a reversible path. Automation and AI assistance must stop safely when authority, freshness, identity, state, or expected outcomes are ambiguous.

Troubleshooting and assurance should locate the first divergence from the expected contract or state machine. Account for time, concurrency, caching, eventual consistency, data and schema drift, partial failure, environment differences, hidden coupling, resource saturation, and missing telemetry. Completion requires reproducible evidence, a causal explanation, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

- Define the objective, owner, authority, scope, assumptions, dependencies, and acceptance criteria for invariants and impossible states.
- Model expected states, boundaries, inputs, outputs, failure modes, and evidence before implementation.
- Validate intended design, implemented behavior, runtime state, and user or mission outcome independently.
- Test positive, negative, boundary, degraded, failed, recovery, scale, and rollback conditions.
- Preserve source, version, configuration, data, telemetry, artifacts, timestamps, decisions, and limitations.
- Automate through typed contracts, least privilege, deterministic gates, bounded concurrency, canaries, and reversibility.

Failure patterns

- Treating invariants and impossible states as a document or tool activity disconnected from actual system behavior.
- Relying on intended design or configuration without proving implementation and runtime outcomes.
- Ignoring failure, concurrency, data, identity, environment, compatibility, and operational boundaries.
- Changing several variables before preserving the original state and stating a falsifiable hypothesis.
- Allowing generated code, tests, findings, or optimizations to proceed without authoritative inputs and review.
- Declaring success after one positive example without negative tests, reproducibility, rollback, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, contracts, data, versions, ownership, and assumptions for invariants and impossible states.
Observe	Runtime state, control flow, telemetry, artifacts, resource behavior, errors, test results, and user-visible outcomes.
Test	Expected, prohibited, boundary, malformed, concurrent, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, resource use, reliability, coverage, risk, maintainability, and operational impact.
Reconcile	Intended requirements and design against code, configuration, runtime behavior, tests, evidence, and residual limitations.

Chapter 02 laboratory and review

Practical laboratory

Refactor string-based errors into typed variants with source chains, API problem details, telemetry, and caller-specific handling.

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 algebraic effects, typed capability failures, proof-carrying results, and compiler-checked recovery completeness.

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

Timeouts, deadlines, cancellation, and resource budgets

Bound work according to end-to-end usefulness rather than arbitrary local timers.

Chapter operating position
Bound work according to end-to-end usefulness rather than arbitrary local timers.

3.1 Deadline propagation

Deadline propagation must be treated as an observable engineering mechanism rather than a slogan, framework checkbox, or tool output. Carry remaining time across calls, queues, retries, databases, workers, and child tasks with explicit ownership. The design should identify authoritative inputs, states, boundaries, assumptions, dependencies, limits, ownership, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the fault, detected condition, typed error, propagation, containment, recovery action, telemetry, and user outcome chain. A diagram, passing test, successful request, exploit result, benchmark, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with implemented state, negative and boundary tests, degraded conditions, raw evidence, recovery, and the resulting user, service, or risk outcome.

Implementation begins with a minimal known-good baseline and explicit invariants. Introduce one variable or dependency at a time, preserve before-state evidence, test expected and prohibited behavior, inject realistic failure, and retain a reversible path. Automation and AI assistance must stop safely when authority, freshness, identity, state, or expected outcomes are ambiguous.

Troubleshooting and assurance should locate the first divergence from the expected contract or state machine. Account for time, concurrency, caching, eventual consistency, data and schema drift, partial failure, environment differences, hidden coupling, resource saturation, and missing telemetry. Completion requires reproducible evidence, a causal explanation, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

- Define the objective, owner, authority, scope, assumptions, dependencies, and acceptance criteria for deadline propagation.
- Model expected states, boundaries, inputs, outputs, failure modes, and evidence before implementation.
- Validate intended design, implemented behavior, runtime state, and user or mission outcome independently.
- Test positive, negative, boundary, degraded, failed, recovery, scale, and rollback conditions.
- Preserve source, version, configuration, data, telemetry, artifacts, timestamps, decisions, and limitations.
- Automate through typed contracts, least privilege, deterministic gates, bounded concurrency, canaries, and reversibility.

Failure patterns

- Treating deadline propagation as a document or tool activity disconnected from actual system behavior.
- Relying on intended design or configuration without proving implementation and runtime outcomes.
- Ignoring failure, concurrency, data, identity, environment, compatibility, and operational boundaries.
- Changing several variables before preserving the original state and stating a falsifiable hypothesis.
- Allowing generated code, tests, findings, or optimizations to proceed without authoritative inputs and review.
- Declaring success after one positive example without negative tests, reproducibility, rollback, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, contracts, data, versions, ownership, and assumptions for deadline propagation.
Observe	Runtime state, control flow, telemetry, artifacts, resource behavior, errors, test results, and user-visible outcomes.
Test	Expected, prohibited, boundary, malformed, concurrent, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, resource use, reliability, coverage, risk, maintainability, and operational impact.
Reconcile	Intended requirements and design against code, configuration, runtime behavior, tests, evidence, and residual limitations.

3.2 Timeout selection and false timeouts

Timeout selection and false timeouts must be treated as an observable engineering mechanism rather than a slogan, framework checkbox, or tool output. Use latency distributions, network variation, queueing, cold starts, contention, and user or workflow objectives. The design should identify authoritative inputs, states, boundaries, assumptions, dependencies, limits, ownership, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the fault, detected condition, typed error, propagation, containment, recovery action, telemetry, and user outcome chain. A diagram, passing test, successful request, exploit result, benchmark, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with implemented state, negative and boundary tests, degraded conditions, raw evidence, recovery, and the resulting user, service, or risk outcome.

Implementation begins with a minimal known-good baseline and explicit invariants. Introduce one variable or dependency at a time, preserve before-state evidence, test expected and prohibited behavior, inject realistic failure, and retain a reversible path. Automation and AI assistance must stop safely when authority, freshness, identity, state, or expected outcomes are ambiguous.

Troubleshooting and assurance should locate the first divergence from the expected contract or state machine. Account for time, concurrency, caching, eventual consistency, data and schema drift, partial failure, environment differences, hidden coupling, resource saturation, and missing telemetry. Completion requires reproducible evidence, a causal explanation, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

- Define the objective, owner, authority, scope, assumptions, dependencies, and acceptance criteria for timeout selection and false timeouts.
- Model expected states, boundaries, inputs, outputs, failure modes, and evidence before implementation.
- Validate intended design, implemented behavior, runtime state, and user or mission outcome independently.
- Test positive, negative, boundary, degraded, failed, recovery, scale, and rollback conditions.
- Preserve source, version, configuration, data, telemetry, artifacts, timestamps, decisions, and limitations.
- Automate through typed contracts, least privilege, deterministic gates, bounded concurrency, canaries, and reversibility.

Implementation sketch

```
error_contract:
  type: dependency-timeout
  retryable: true
  operation_outcome: unknown
  retry_policy:
    max_attempts: 2
    budget_from_parent_deadline: true
    jitter: full
  idempotency_key: required
  telemetry: [dependency, elapsed, deadline, attempt, effect_state]
  fallback: explicit-degraded-response
```

Failure patterns

- Treating timeout selection and false timeouts as a document or tool activity disconnected from actual system behavior.
- Relying on intended design or configuration without proving implementation and runtime outcomes.
- Ignoring failure, concurrency, data, identity, environment, compatibility, and operational boundaries.

Changing several variables before preserving the original state and stating a falsifiable hypothesis.

Allowing generated code, tests, findings, or optimizations to proceed without authoritative inputs and review.

Declaring success after one positive example without negative tests, reproducibility, rollback, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, contracts, data, versions, ownership, and assumptions for timeout selection and false timeouts.
Observe	Runtime state, control flow, telemetry, artifacts, resource behavior, errors, test results, and user-visible outcomes.
Test	Expected, prohibited, boundary, malformed, concurrent, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, resource use, reliability, coverage, risk, maintainability, and operational impact.
Reconcile	Intended requirements and design against code, configuration, runtime behavior, tests, evidence, and residual limitations.

3.3 Cancellation and cleanup

Cancellation and cleanup must be treated as an observable engineering mechanism rather than a slogan, framework checkbox, or tool output. Stop child work, release resources, roll back or compensate, preserve state, and avoid cancellation-unsafe critical sections. The design should identify authoritative inputs, states, boundaries, assumptions, dependencies, limits, ownership, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the fault, detected condition, typed error, propagation, containment, recovery action, telemetry, and user outcome chain. A diagram, passing test, successful request, exploit result, benchmark, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with implemented state, negative and boundary tests, degraded conditions, raw evidence, recovery, and the resulting user, service, or risk outcome.

Implementation begins with a minimal known-good baseline and explicit invariants. Introduce one variable or dependency at a time, preserve before-state evidence, test expected and prohibited behavior, inject realistic failure, and retain a reversible path. Automation and AI assistance must stop safely when authority, freshness, identity, state, or expected outcomes are ambiguous.

Troubleshooting and assurance should locate the first divergence from the expected contract or state machine. Account for time, concurrency, caching, eventual consistency, data and schema drift, partial failure, environment differences, hidden coupling, resource saturation, and missing telemetry. Completion requires reproducible evidence, a causal explanation, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

Define the objective, owner, authority, scope, assumptions, dependencies, and acceptance criteria for cancellation and cleanup.

Model expected states, boundaries, inputs, outputs, failure modes, and evidence before implementation.

Validate intended design, implemented behavior, runtime state, and user or mission outcome independently.

Test positive, negative, boundary, degraded, failed, recovery, scale, and rollback conditions.

Preserve source, version, configuration, data, telemetry, artifacts, timestamps, decisions, and limitations.

Automate through typed contracts, least privilege, deterministic gates, bounded concurrency, canaries, and reversibility.

Failure patterns

Treating cancellation and cleanup as a document or tool activity disconnected from actual system behavior.

Relying on intended design or configuration without proving implementation and runtime outcomes.

Ignoring failure, concurrency, data, identity, environment, compatibility, and operational boundaries.

Changing several variables before preserving the original state and stating a falsifiable hypothesis.

Allowing generated code, tests, findings, or optimizations to proceed without authoritative inputs and review.

Declaring success after one positive example without negative tests, reproducibility, rollback, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, contracts, data, versions, ownership, and assumptions for cancellation and cleanup.
Observe	Runtime state, control flow, telemetry, artifacts, resource behavior, errors, test results, and user-visible outcomes.
Test	Expected, prohibited, boundary, malformed, concurrent, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, resource use, reliability, coverage, risk, maintainability, and operational impact.
Reconcile	Intended requirements and design against code, configuration, runtime behavior, tests, evidence, and residual limitations.

3.4 Resource budgets

Resource budgets must be treated as an observable engineering mechanism rather than a slogan, framework checkbox, or tool output. Allocate connections, threads, tasks, memory, CPU, file descriptors, queue capacity, and external quota across work. The design should identify authoritative inputs, states, boundaries, assumptions, dependencies, limits, ownership, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the fault, detected condition, typed error, propagation, containment, recovery action, telemetry, and user outcome chain. A diagram, passing test, successful request, exploit result, benchmark, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with implemented state, negative and boundary tests, degraded conditions, raw evidence, recovery, and the resulting user, service, or risk outcome.

Implementation begins with a minimal known-good baseline and explicit invariants. Introduce one variable or dependency at a time, preserve before-state evidence, test expected and prohibited behavior, inject realistic failure, and retain a reversible path. Automation and AI assistance must stop safely when authority, freshness, identity, state, or expected outcomes are ambiguous.

Troubleshooting and assurance should locate the first divergence from the expected contract or state machine. Account for time, concurrency, caching, eventual consistency, data and schema drift, partial failure, environment differences, hidden coupling, resource saturation, and missing telemetry. Completion requires reproducible evidence, a causal explanation, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

- Define the objective, owner, authority, scope, assumptions, dependencies, and acceptance criteria for resource budgets.
- Model expected states, boundaries, inputs, outputs, failure modes, and evidence before implementation.
- Validate intended design, implemented behavior, runtime state, and user or mission outcome independently.
- Test positive, negative, boundary, degraded, failed, recovery, scale, and rollback conditions.
- Preserve source, version, configuration, data, telemetry, artifacts, timestamps, decisions, and limitations.
- Automate through typed contracts, least privilege, deterministic gates, bounded concurrency, canaries, and reversibility.

Failure patterns

- Treating resource budgets as a document or tool activity disconnected from actual system behavior.
- Relying on intended design or configuration without proving implementation and runtime outcomes.
- Ignoring failure, concurrency, data, identity, environment, compatibility, and operational boundaries.
- Changing several variables before preserving the original state and stating a falsifiable hypothesis.
- Allowing generated code, tests, findings, or optimizations to proceed without authoritative inputs and review.
- Declaring success after one positive example without negative tests, reproducibility, rollback, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, contracts, data, versions, ownership, and assumptions for resource budgets.
Observe	Runtime state, control flow, telemetry, artifacts, resource behavior, errors, test results, and user-visible outcomes.
Test	Expected, prohibited, boundary, malformed, concurrent, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, resource use, reliability, coverage, risk, maintainability, and operational impact.
Reconcile	Intended requirements and design against code, configuration, runtime behavior, tests, evidence, and residual limitations.

Chapter 03 laboratory and review

Practical laboratory
Implement an end-to-end deadline across three simulated services and verify cancellation, cleanup, partial completion, and telemetry.

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 budget-aware scheduling, distributed cancellation tokens, and agent plans constrained by time, cost, and risk.

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

Retries, idempotency, circuit breakers, and backpressure

Prevent recovery mechanisms from amplifying failures or duplicating effects.

Chapter operating position
Prevent recovery mechanisms from amplifying failures or duplicating effects.

4.1 Retry safety and policy

Retry safety and policy must be treated as an observable engineering mechanism rather than a slogan, framework checkbox, or tool output. Retry only known transient conditions with capped attempts, jittered backoff, deadlines, budgets, and operation semantics. The design should identify authoritative inputs, states, boundaries, assumptions, dependencies, limits, ownership, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the fault, detected condition, typed error, propagation, containment, recovery action, telemetry, and user outcome chain. A diagram, passing test, successful request, exploit result, benchmark, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with implemented state, negative and boundary tests, degraded conditions, raw evidence, recovery, and the resulting user, service, or risk outcome.

Implementation begins with a minimal known-good baseline and explicit invariants. Introduce one variable or dependency at a time, preserve before-state evidence, test expected and prohibited behavior, inject realistic failure, and retain a reversible path. Automation and AI assistance must stop safely when authority, freshness, identity, state, or expected outcomes are ambiguous.

Troubleshooting and assurance should locate the first divergence from the expected contract or state machine. Account for time, concurrency, caching, eventual consistency, data and schema drift, partial failure, environment differences, hidden coupling, resource saturation, and missing telemetry. Completion requires reproducible evidence, a causal explanation, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

- Define the objective, owner, authority, scope, assumptions, dependencies, and acceptance criteria for retry safety and policy.
- Model expected states, boundaries, inputs, outputs, failure modes, and evidence before implementation.
- Validate intended design, implemented behavior, runtime state, and user or mission outcome independently.
- Test positive, negative, boundary, degraded, failed, recovery, scale, and rollback conditions.
- Preserve source, version, configuration, data, telemetry, artifacts, timestamps, decisions, and limitations.
- Automate through typed contracts, least privilege, deterministic gates, bounded concurrency, canaries, and reversibility.

Failure patterns

- Treating retry safety and policy as a document or tool activity disconnected from actual system behavior.
- Relying on intended design or configuration without proving implementation and runtime outcomes.
- Ignoring failure, concurrency, data, identity, environment, compatibility, and operational boundaries.
- Changing several variables before preserving the original state and stating a falsifiable hypothesis.
- Allowing generated code, tests, findings, or optimizations to proceed without authoritative inputs and review.
- Declaring success after one positive example without negative tests, reproducibility, rollback, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, contracts, data, versions, ownership, and assumptions for retry safety and policy.
Observe	Runtime state, control flow, telemetry, artifacts, resource behavior, errors, test results, and user-visible outcomes.
Test	Expected, prohibited, boundary, malformed, concurrent, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, resource use, reliability, coverage, risk, maintainability, and operational impact.
Reconcile	Intended requirements and design against code, configuration, runtime behavior, tests, evidence, and residual limitations.

4.2 Idempotency and deduplication

Idempotency and deduplication must be treated as an observable engineering mechanism rather than a slogan, framework checkbox, or tool output. Use operation identity, keys, state, stored results, sequence, replay windows, and effect reconciliation. The design should identify authoritative inputs, states, boundaries, assumptions, dependencies, limits, ownership, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the fault, detected condition, typed error, propagation, containment, recovery action, telemetry, and user outcome chain. A diagram, passing test, successful request, exploit result, benchmark, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with implemented state, negative and boundary tests, degraded conditions, raw evidence, recovery, and the resulting user, service, or risk outcome.

Implementation begins with a minimal known-good baseline and explicit invariants. Introduce one variable or dependency at a time, preserve before-state evidence, test expected and prohibited behavior, inject realistic failure, and retain a reversible path. Automation and AI assistance must stop safely when authority, freshness, identity, state, or expected outcomes are ambiguous.

Troubleshooting and assurance should locate the first divergence from the expected contract or state machine. Account for time, concurrency, caching, eventual consistency, data and schema drift, partial failure, environment differences, hidden coupling, resource saturation, and missing telemetry. Completion requires reproducible evidence, a causal explanation, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

- Define the objective, owner, authority, scope, assumptions, dependencies, and acceptance criteria for idempotency and deduplication.
- Model expected states, boundaries, inputs, outputs, failure modes, and evidence before implementation.
- Validate intended design, implemented behavior, runtime state, and user or mission outcome independently.
- Test positive, negative, boundary, degraded, failed, recovery, scale, and rollback conditions.
- Preserve source, version, configuration, data, telemetry, artifacts, timestamps, decisions, and limitations.
- Automate through typed contracts, least privilege, deterministic gates, bounded concurrency, canaries, and reversibility.

Implementation sketch

```
error_contract:
  type: dependency-timeout
  retryable: true
  operation_outcome: unknown
  retry_policy:
    max_attempts: 2
    budget_from_parent_deadline: true
    jitter: full
  idempotency_key: required
  telemetry: [dependency, elapsed, deadline, attempt, effect_state]
  fallback: explicit-degraded-response
```

Failure patterns

- Treating idempotency and deduplication as a document or tool activity disconnected from actual system behavior.
- Relying on intended design or configuration without proving implementation and runtime outcomes.
- Ignoring failure, concurrency, data, identity, environment, compatibility, and operational boundaries.

Changing several variables before preserving the original state and stating a falsifiable hypothesis.

Allowing generated code, tests, findings, or optimizations to proceed without authoritative inputs and review.

Declaring success after one positive example without negative tests, reproducibility, rollback, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, contracts, data, versions, ownership, and assumptions for idempotency and deduplication.
Observe	Runtime state, control flow, telemetry, artifacts, resource behavior, errors, test results, and user-visible outcomes.
Test	Expected, prohibited, boundary, malformed, concurrent, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, resource use, reliability, coverage, risk, maintainability, and operational impact.
Reconcile	Intended requirements and design against code, configuration, runtime behavior, tests, evidence, and residual limitations.

4.3 Circuit breakers and health

Circuit breakers and health must be treated as an observable engineering mechanism rather than a slogan, framework checkbox, or tool output. Open, probe, and close according to meaningful dependency health, request classes, stale data, and fallback capacity. The design should identify authoritative inputs, states, boundaries, assumptions, dependencies, limits, ownership, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the fault, detected condition, typed error, propagation, containment, recovery action, telemetry, and user outcome chain. A diagram, passing test, successful request, exploit result, benchmark, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with implemented state, negative and boundary tests, degraded conditions, raw evidence, recovery, and the resulting user, service, or risk outcome.

Implementation begins with a minimal known-good baseline and explicit invariants. Introduce one variable or dependency at a time, preserve before-state evidence, test expected and prohibited behavior, inject realistic failure, and retain a reversible path. Automation and AI assistance must stop safely when authority, freshness, identity, state, or expected outcomes are ambiguous.

Troubleshooting and assurance should locate the first divergence from the expected contract or state machine. Account for time, concurrency, caching, eventual consistency, data and schema drift, partial failure, environment differences, hidden coupling, resource saturation, and missing telemetry. Completion requires reproducible evidence, a causal explanation, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

Define the objective, owner, authority, scope, assumptions, dependencies, and acceptance criteria for circuit breakers and health.

Model expected states, boundaries, inputs, outputs, failure modes, and evidence before implementation.

Validate intended design, implemented behavior, runtime state, and user or mission outcome independently.

Test positive, negative, boundary, degraded, failed, recovery, scale, and rollback conditions.

Preserve source, version, configuration, data, telemetry, artifacts, timestamps, decisions, and limitations.

Automate through typed contracts, least privilege, deterministic gates, bounded concurrency, canaries, and reversibility.

Failure patterns

Treating circuit breakers and health as a document or tool activity disconnected from actual system behavior.

Relying on intended design or configuration without proving implementation and runtime outcomes.

Ignoring failure, concurrency, data, identity, environment, compatibility, and operational boundaries.

Changing several variables before preserving the original state and stating a falsifiable hypothesis.

Allowing generated code, tests, findings, or optimizations to proceed without authoritative inputs and review.

Declaring success after one positive example without negative tests, reproducibility, rollback, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, contracts, data, versions, ownership, and assumptions for circuit breakers and health.
Observe	Runtime state, control flow, telemetry, artifacts, resource behavior, errors, test results, and user-visible outcomes.
Test	Expected, prohibited, boundary, malformed, concurrent, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, resource use, reliability, coverage, risk, maintainability, and operational impact.
Reconcile	Intended requirements and design against code, configuration, runtime behavior, tests, evidence, and residual limitations.

4.4 Backpressure and admission control

Backpressure and admission control must be treated as an observable engineering mechanism rather than a slogan, framework checkbox, or tool output. Bound queues, reject early, prioritize, shed load, slow producers, and preserve critical work during overload. The design should identify authoritative inputs, states, boundaries, assumptions, dependencies, limits, ownership, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the fault, detected condition, typed error, propagation, containment, recovery action, telemetry, and user outcome chain. A diagram, passing test, successful request, exploit result, benchmark, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with implemented state, negative and boundary tests, degraded conditions, raw evidence, recovery, and the resulting user, service, or risk outcome.

Implementation begins with a minimal known-good baseline and explicit invariants. Introduce one variable or dependency at a time, preserve before-state evidence, test expected and prohibited behavior, inject realistic failure, and retain a reversible path. Automation and AI assistance must stop safely when authority, freshness, identity, state, or expected outcomes are ambiguous.

Troubleshooting and assurance should locate the first divergence from the expected contract or state machine. Account for time, concurrency, caching, eventual consistency, data and schema drift, partial failure, environment differences, hidden coupling, resource saturation, and missing telemetry. Completion requires reproducible evidence, a causal explanation, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

- Define the objective, owner, authority, scope, assumptions, dependencies, and acceptance criteria for backpressure and admission control.
- Model expected states, boundaries, inputs, outputs, failure modes, and evidence before implementation.
- Validate intended design, implemented behavior, runtime state, and user or mission outcome independently.
- Test positive, negative, boundary, degraded, failed, recovery, scale, and rollback conditions.
- Preserve source, version, configuration, data, telemetry, artifacts, timestamps, decisions, and limitations.
- Automate through typed contracts, least privilege, deterministic gates, bounded concurrency, canaries, and reversibility.

Failure patterns

- Treating backpressure and admission control as a document or tool activity disconnected from actual system behavior.
- Relying on intended design or configuration without proving implementation and runtime outcomes.
- Ignoring failure, concurrency, data, identity, environment, compatibility, and operational boundaries.
- Changing several variables before preserving the original state and stating a falsifiable hypothesis.
- Allowing generated code, tests, findings, or optimizations to proceed without authoritative inputs and review.
- Declaring success after one positive example without negative tests, reproducibility, rollback, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, contracts, data, versions, ownership, and assumptions for backpressure and admission control.
Observe	Runtime state, control flow, telemetry, artifacts, resource behavior, errors, test results, and user-visible outcomes.
Test	Expected, prohibited, boundary, malformed, concurrent, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, resource use, reliability, coverage, risk, maintainability, and operational impact.
Reconcile	Intended requirements and design against code, configuration, runtime behavior, tests, evidence, and residual limitations.

Chapter 04 laboratory and review

Practical laboratory
Create a retry storm in a lab, then stabilize it with deadlines, budgets, backoff, idempotency, circuit breaking, and admission control.

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 concurrency limits, predictive shedding, and formal retry-budget 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

Partial failure, consistency, and workflow recovery

Define what completion means when multiple steps, services, or data stores are involved.

Chapter operating position
Define what completion means when multiple steps, services, or data stores are involved.

5.1 Atomic and non-atomic boundaries

Atomic and non-atomic boundaries must be treated as an observable engineering mechanism rather than a slogan, framework checkbox, or tool output. Identify transaction scope, external side effects, durable points, visibility, and irreversible actions. The design should identify authoritative inputs, states, boundaries, assumptions, dependencies, limits, ownership, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the fault, detected condition, typed error, propagation, containment, recovery action, telemetry, and user outcome chain. A diagram, passing test, successful request, exploit result, benchmark, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with implemented state, negative and boundary tests, degraded conditions, raw evidence, recovery, and the resulting user, service, or risk outcome.

Implementation begins with a minimal known-good baseline and explicit invariants. Introduce one variable or dependency at a time, preserve before-state evidence, test expected and prohibited behavior, inject realistic failure, and retain a reversible path. Automation and AI assistance must stop safely when authority, freshness, identity, state, or expected outcomes are ambiguous.

Troubleshooting and assurance should locate the first divergence from the expected contract or state machine. Account for time, concurrency, caching, eventual consistency, data and schema drift, partial failure, environment differences, hidden coupling, resource saturation, and missing telemetry. Completion requires reproducible evidence, a causal explanation, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

- Define the objective, owner, authority, scope, assumptions, dependencies, and acceptance criteria for atomic and non-atomic boundaries.
- Model expected states, boundaries, inputs, outputs, failure modes, and evidence before implementation.
- Validate intended design, implemented behavior, runtime state, and user or mission outcome independently.
- Test positive, negative, boundary, degraded, failed, recovery, scale, and rollback conditions.
- Preserve source, version, configuration, data, telemetry, artifacts, timestamps, decisions, and limitations.
- Automate through typed contracts, least privilege, deterministic gates, bounded concurrency, canaries, and reversibility.

Failure patterns

- Treating atomic and non-atomic boundaries as a document or tool activity disconnected from actual system behavior.
- Relying on intended design or configuration without proving implementation and runtime outcomes.
- Ignoring failure, concurrency, data, identity, environment, compatibility, and operational boundaries.
- Changing several variables before preserving the original state and stating a falsifiable hypothesis.
- Allowing generated code, tests, findings, or optimizations to proceed without authoritative inputs and review.
- Declaring success after one positive example without negative tests, reproducibility, rollback, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, contracts, data, versions, ownership, and assumptions for atomic and non-atomic boundaries.
Observe	Runtime state, control flow, telemetry, artifacts, resource behavior, errors, test results, and user-visible outcomes.
Test	Expected, prohibited, boundary, malformed, concurrent, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, resource use, reliability, coverage, risk, maintainability, and operational impact.
Reconcile	Intended requirements and design against code, configuration, runtime behavior, tests, evidence, and residual limitations.

5.2 Sagas and compensation

Sagas and compensation must be treated as an observable engineering mechanism rather than a slogan, framework checkbox, or tool output. Design local transactions, pivots, retryable steps, compensations, semantic undo, and manual resolution. The design should identify authoritative inputs, states, boundaries, assumptions, dependencies, limits, ownership, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the fault, detected condition, typed error, propagation, containment, recovery action, telemetry, and user outcome chain. A diagram, passing test, successful request, exploit result, benchmark, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with implemented state, negative and boundary tests, degraded conditions, raw evidence, recovery, and the resulting user, service, or risk outcome.

Implementation begins with a minimal known-good baseline and explicit invariants. Introduce one variable or dependency at a time, preserve before-state evidence, test expected and prohibited behavior, inject realistic failure, and retain a reversible path. Automation and AI assistance must stop safely when authority, freshness, identity, state, or expected outcomes are ambiguous.

Troubleshooting and assurance should locate the first divergence from the expected contract or state machine. Account for time, concurrency, caching, eventual consistency, data and schema drift, partial failure, environment differences, hidden coupling, resource saturation, and missing telemetry. Completion requires reproducible evidence, a causal explanation, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

- Define the objective, owner, authority, scope, assumptions, dependencies, and acceptance criteria for sagas and compensation.
- Model expected states, boundaries, inputs, outputs, failure modes, and evidence before implementation.
- Validate intended design, implemented behavior, runtime state, and user or mission outcome independently.
- Test positive, negative, boundary, degraded, failed, recovery, scale, and rollback conditions.
- Preserve source, version, configuration, data, telemetry, artifacts, timestamps, decisions, and limitations.
- Automate through typed contracts, least privilege, deterministic gates, bounded concurrency, canaries, and reversibility.

Implementation sketch

```
error_contract:
  type: dependency-timeout
  retryable: true
  operation_outcome: unknown
  retry_policy:
    max_attempts: 2
    budget_from_parent_deadline: true
    jitter: full
  idempotency_key: required
  telemetry: [dependency, elapsed, deadline, attempt, effect_state]
  fallback: explicit-degraded-response
```

Failure patterns

- Treating sagas and compensation as a document or tool activity disconnected from actual system behavior.
- Relying on intended design or configuration without proving implementation and runtime outcomes.

Ignoring failure, concurrency, data, identity, environment, compatibility, and operational boundaries.

Changing several variables before preserving the original state and stating a falsifiable hypothesis.

Allowing generated code, tests, findings, or optimizations to proceed without authoritative inputs and review.

Declaring success after one positive example without negative tests, reproducibility, rollback, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, contracts, data, versions, ownership, and assumptions for sagas and compensation.
Observe	Runtime state, control flow, telemetry, artifacts, resource behavior, errors, test results, and user-visible outcomes.
Test	Expected, prohibited, boundary, malformed, concurrent, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, resource use, reliability, coverage, risk, maintainability, and operational impact.
Reconcile	Intended requirements and design against code, configuration, runtime behavior, tests, evidence, and residual limitations.

5.3 Reconciliation and repair

Reconciliation and repair must be treated as an observable engineering mechanism rather than a slogan, framework checkbox, or tool output. Detect divergence, compare authority, replay, rebuild, correct, quarantine, and document uncertain state. The design should identify authoritative inputs, states, boundaries, assumptions, dependencies, limits, ownership, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the fault, detected condition, typed error, propagation, containment, recovery action, telemetry, and user outcome chain. A diagram, passing test, successful request, exploit result, benchmark, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with implemented state, negative and boundary tests, degraded conditions, raw evidence, recovery, and the resulting user, service, or risk outcome.

Implementation begins with a minimal known-good baseline and explicit invariants. Introduce one variable or dependency at a time, preserve before-state evidence, test expected and prohibited behavior, inject realistic failure, and retain a reversible path. Automation and AI assistance must stop safely when authority, freshness, identity, state, or expected outcomes are ambiguous.

Troubleshooting and assurance should locate the first divergence from the expected contract or state machine. Account for time, concurrency, caching, eventual consistency, data and schema drift, partial failure, environment differences, hidden coupling, resource saturation, and missing telemetry. Completion requires reproducible evidence, a causal explanation, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

Define the objective, owner, authority, scope, assumptions, dependencies, and acceptance criteria for reconciliation and repair.

Model expected states, boundaries, inputs, outputs, failure modes, and evidence before implementation.

Validate intended design, implemented behavior, runtime state, and user or mission outcome independently.

Test positive, negative, boundary, degraded, failed, recovery, scale, and rollback conditions.

Preserve source, version, configuration, data, telemetry, artifacts, timestamps, decisions, and limitations.

Automate through typed contracts, least privilege, deterministic gates, bounded concurrency, canaries, and reversibility.

Failure patterns

Treating reconciliation and repair as a document or tool activity disconnected from actual system behavior.

Relying on intended design or configuration without proving implementation and runtime outcomes.

Ignoring failure, concurrency, data, identity, environment, compatibility, and operational boundaries.

Changing several variables before preserving the original state and stating a falsifiable hypothesis.

Allowing generated code, tests, findings, or optimizations to proceed without authoritative inputs and review.

Declaring success after one positive example without negative tests, reproducibility, rollback, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, contracts, data, versions, ownership, and assumptions for reconciliation and repair.
Observe	Runtime state, control flow, telemetry, artifacts, resource behavior, errors, test results, and user-visible outcomes.
Test	Expected, prohibited, boundary, malformed, concurrent, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, resource use, reliability, coverage, risk, maintainability, and operational impact.
Reconcile	Intended requirements and design against code, configuration, runtime behavior, tests, evidence, and residual limitations.

5.4 Exactly-once claims and practical effects

Exactly-once claims and practical effects must be treated as an observable engineering mechanism rather than a slogan, framework checkbox, or tool output. Separate delivery from effect, use idempotency and state, and disclose remaining duplicate or loss conditions. The design should identify authoritative inputs, states, boundaries, assumptions, dependencies, limits, ownership, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the fault, detected condition, typed error, propagation, containment, recovery action, telemetry, and user outcome chain. A diagram, passing test, successful request, exploit result, benchmark, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with implemented state, negative and boundary tests, degraded conditions, raw evidence, recovery, and the resulting user, service, or risk outcome.

Implementation begins with a minimal known-good baseline and explicit invariants. Introduce one variable or dependency at a time, preserve before-state evidence, test expected and prohibited behavior, inject realistic failure, and retain a reversible path. Automation and AI assistance must stop safely when authority, freshness, identity, state, or expected outcomes are ambiguous.

Troubleshooting and assurance should locate the first divergence from the expected contract or state machine. Account for time, concurrency, caching, eventual consistency, data and schema drift, partial failure, environment differences, hidden coupling, resource saturation, and missing telemetry. Completion requires reproducible evidence, a causal explanation, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

- Define the objective, owner, authority, scope, assumptions, dependencies, and acceptance criteria for exactly-once claims and practical effects.
- Model expected states, boundaries, inputs, outputs, failure modes, and evidence before implementation.
- Validate intended design, implemented behavior, runtime state, and user or mission outcome independently.
- Test positive, negative, boundary, degraded, failed, recovery, scale, and rollback conditions.
- Preserve source, version, configuration, data, telemetry, artifacts, timestamps, decisions, and limitations.
- Automate through typed contracts, least privilege, deterministic gates, bounded concurrency, canaries, and reversibility.

Failure patterns

- Treating exactly-once claims and practical effects as a document or tool activity disconnected from actual system behavior.
- Relying on intended design or configuration without proving implementation and runtime outcomes.
- Ignoring failure, concurrency, data, identity, environment, compatibility, and operational boundaries.
- Changing several variables before preserving the original state and stating a falsifiable hypothesis.
- Allowing generated code, tests, findings, or optimizations to proceed without authoritative inputs and review.
- Declaring success after one positive example without negative tests, reproducibility, rollback, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, contracts, data, versions, ownership, and assumptions for exactly-once claims and practical effects.
Observe	Runtime state, control flow, telemetry, artifacts, resource behavior, errors, test results, and user-visible outcomes.
Test	Expected, prohibited, boundary, malformed, concurrent, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, resource use, reliability, coverage, risk, maintainability, and operational impact.
Reconcile	Intended requirements and design against code, configuration, runtime behavior, tests, evidence, and residual limitations.

Chapter 05 laboratory and review

Practical laboratory
Implement a multi-step workflow with a pivot, injected failures, compensations, retries, and a reconciliation process for uncertain outcomes.

Laboratory evidence package

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

Frontier extension

Explore verifiable workflow journals, deterministic replay, and autonomous repair with human exception authority.

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

Graceful degradation, isolation, and recovery

Preserve essential service and prevent local faults from becoming system-wide failure.

Chapter operating position
Preserve essential service and prevent local faults from becoming system-wide failure.

6.1 Bulkheads and failure domains

Bulkheads and failure domains must be treated as an observable engineering mechanism rather than a slogan, framework checkbox, or tool output. Partition resources, tenants, workloads, dependencies, queues, regions, and administrative authority. The design should identify authoritative inputs, states, boundaries, assumptions, dependencies, limits, ownership, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the fault, detected condition, typed error, propagation, containment, recovery action, telemetry, and user outcome chain. A diagram, passing test, successful request, exploit result, benchmark, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with implemented state, negative and boundary tests, degraded conditions, raw evidence, recovery, and the resulting user, service, or risk outcome.

Implementation begins with a minimal known-good baseline and explicit invariants. Introduce one variable or dependency at a time, preserve before-state evidence, test expected and prohibited behavior, inject realistic failure, and retain a reversible path. Automation and AI assistance must stop safely when authority, freshness, identity, state, or expected outcomes are ambiguous.

Troubleshooting and assurance should locate the first divergence from the expected contract or state machine. Account for time, concurrency, caching, eventual consistency, data and schema drift, partial failure, environment differences, hidden coupling, resource saturation, and missing telemetry. Completion requires reproducible evidence, a causal explanation, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

- Define the objective, owner, authority, scope, assumptions, dependencies, and acceptance criteria for bulkheads and failure domains.
- Model expected states, boundaries, inputs, outputs, failure modes, and evidence before implementation.
- Validate intended design, implemented behavior, runtime state, and user or mission outcome independently.
- Test positive, negative, boundary, degraded, failed, recovery, scale, and rollback conditions.
- Preserve source, version, configuration, data, telemetry, artifacts, timestamps, decisions, and limitations.
- Automate through typed contracts, least privilege, deterministic gates, bounded concurrency, canaries, and reversibility.

Failure patterns

- Treating bulkheads and failure domains as a document or tool activity disconnected from actual system behavior.
- Relying on intended design or configuration without proving implementation and runtime outcomes.
- Ignoring failure, concurrency, data, identity, environment, compatibility, and operational boundaries.
- Changing several variables before preserving the original state and stating a falsifiable hypothesis.
- Allowing generated code, tests, findings, or optimizations to proceed without authoritative inputs and review.
- Declaring success after one positive example without negative tests, reproducibility, rollback, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, contracts, data, versions, ownership, and assumptions for bulkheads and failure domains.
Observe	Runtime state, control flow, telemetry, artifacts, resource behavior, errors, test results, and user-visible outcomes.
Test	Expected, prohibited, boundary, malformed, concurrent, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, resource use, reliability, coverage, risk, maintainability, and operational impact.
Reconcile	Intended requirements and design against code, configuration, runtime behavior, tests, evidence, and residual limitations.

6.2 Fallback and degraded modes

Fallback and degraded modes must be treated as an observable engineering mechanism rather than a slogan, framework checkbox, or tool output. Use cached, stale, partial, reduced, read-only, offline, manual, and alternate-service modes with clear semantics. The design should identify authoritative inputs, states, boundaries, assumptions, dependencies, limits, ownership, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the fault, detected condition, typed error, propagation, containment, recovery action, telemetry, and user outcome chain. A diagram, passing test, successful request, exploit result, benchmark, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with implemented state, negative and boundary tests, degraded conditions, raw evidence, recovery, and the resulting user, service, or risk outcome.

Implementation begins with a minimal known-good baseline and explicit invariants. Introduce one variable or dependency at a time, preserve before-state evidence, test expected and prohibited behavior, inject realistic failure, and retain a reversible path. Automation and AI assistance must stop safely when authority, freshness, identity, state, or expected outcomes are ambiguous.

Troubleshooting and assurance should locate the first divergence from the expected contract or state machine. Account for time, concurrency, caching, eventual consistency, data and schema drift, partial failure, environment differences, hidden coupling, resource saturation, and missing telemetry. Completion requires reproducible evidence, a causal explanation, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

- Define the objective, owner, authority, scope, assumptions, dependencies, and acceptance criteria for fallback and degraded modes.
- Model expected states, boundaries, inputs, outputs, failure modes, and evidence before implementation.
- Validate intended design, implemented behavior, runtime state, and user or mission outcome independently.
- Test positive, negative, boundary, degraded, failed, recovery, scale, and rollback conditions.
- Preserve source, version, configuration, data, telemetry, artifacts, timestamps, decisions, and limitations.
- Automate through typed contracts, least privilege, deterministic gates, bounded concurrency, canaries, and reversibility.

Implementation sketch

```
error_contract:
  type: dependency-timeout
  retryable: true
  operation_outcome: unknown
  retry_policy:
    max_attempts: 2
    budget_from_parent_deadline: true
    jitter: full
  idempotency_key: required
  telemetry: [dependency, elapsed, deadline, attempt, effect_state]
  fallback: explicit-degraded-response
```

Failure patterns

- Treating fallback and degraded modes as a document or tool activity disconnected from actual system behavior.
- Relying on intended design or configuration without proving implementation and runtime outcomes.

Ignoring failure, concurrency, data, identity, environment, compatibility, and operational boundaries.

Changing several variables before preserving the original state and stating a falsifiable hypothesis.

Allowing generated code, tests, findings, or optimizations to proceed without authoritative inputs and review.

Declaring success after one positive example without negative tests, reproducibility, rollback, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, contracts, data, versions, ownership, and assumptions for fallback and degraded modes.
Observe	Runtime state, control flow, telemetry, artifacts, resource behavior, errors, test results, and user-visible outcomes.
Test	Expected, prohibited, boundary, malformed, concurrent, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, resource use, reliability, coverage, risk, maintainability, and operational impact.
Reconcile	Intended requirements and design against code, configuration, runtime behavior, tests, evidence, and residual limitations.

6.3 Fail-safe and fail-secure behavior

Fail-safe and fail-secure behavior must be treated as an observable engineering mechanism rather than a slogan, framework checkbox, or tool output. Choose availability, safety, privacy, integrity, authorization, and data-preservation outcomes deliberately. The design should identify authoritative inputs, states, boundaries, assumptions, dependencies, limits, ownership, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the fault, detected condition, typed error, propagation, containment, recovery action, telemetry, and user outcome chain. A diagram, passing test, successful request, exploit result, benchmark, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with implemented state, negative and boundary tests, degraded conditions, raw evidence, recovery, and the resulting user, service, or risk outcome.

Implementation begins with a minimal known-good baseline and explicit invariants. Introduce one variable or dependency at a time, preserve before-state evidence, test expected and prohibited behavior, inject realistic failure, and retain a reversible path. Automation and AI assistance must stop safely when authority, freshness, identity, state, or expected outcomes are ambiguous.

Troubleshooting and assurance should locate the first divergence from the expected contract or state machine. Account for time, concurrency, caching, eventual consistency, data and schema drift, partial failure, environment differences, hidden coupling, resource saturation, and missing telemetry. Completion requires reproducible evidence, a causal explanation, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

Define the objective, owner, authority, scope, assumptions, dependencies, and acceptance criteria for fail-safe and fail-secure behavior.

Model expected states, boundaries, inputs, outputs, failure modes, and evidence before implementation.

Validate intended design, implemented behavior, runtime state, and user or mission outcome independently.

Test positive, negative, boundary, degraded, failed, recovery, scale, and rollback conditions.

Preserve source, version, configuration, data, telemetry, artifacts, timestamps, decisions, and limitations.

Automate through typed contracts, least privilege, deterministic gates, bounded concurrency, canaries, and reversibility.

Failure patterns

Treating fail-safe and fail-secure behavior as a document or tool activity disconnected from actual system behavior.

Relying on intended design or configuration without proving implementation and runtime outcomes.

Ignoring failure, concurrency, data, identity, environment, compatibility, and operational boundaries.

Changing several variables before preserving the original state and stating a falsifiable hypothesis.

Allowing generated code, tests, findings, or optimizations to proceed without authoritative inputs and review.

Declaring success after one positive example without negative tests, reproducibility, rollback, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, contracts, data, versions, ownership, and assumptions for fail-safe and fail-secure behavior.
Observe	Runtime state, control flow, telemetry, artifacts, resource behavior, errors, test results, and user-visible outcomes.
Test	Expected, prohibited, boundary, malformed, concurrent, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, resource use, reliability, coverage, risk, maintainability, and operational impact.
Reconcile	Intended requirements and design against code, configuration, runtime behavior, tests, evidence, and residual limitations.

6.4 Restart, rollback, and recovery

Restart, rollback, and recovery must be treated as an observable engineering mechanism rather than a slogan, framework checkbox, or tool output. Use supervisors, checkpoints, snapshots, immutable deploys, rebuild, data recovery, and validated return to service. The design should identify authoritative inputs, states, boundaries, assumptions, dependencies, limits, ownership, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the fault, detected condition, typed error, propagation, containment, recovery action, telemetry, and user outcome chain. A diagram, passing test, successful request, exploit result, benchmark, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with implemented state, negative and boundary tests, degraded conditions, raw evidence, recovery, and the resulting user, service, or risk outcome.

Implementation begins with a minimal known-good baseline and explicit invariants. Introduce one variable or dependency at a time, preserve before-state evidence, test expected and prohibited behavior, inject realistic failure, and retain a reversible path. Automation and AI assistance must stop safely when authority, freshness, identity, state, or expected outcomes are ambiguous.

Troubleshooting and assurance should locate the first divergence from the expected contract or state machine. Account for time, concurrency, caching, eventual consistency, data and schema drift, partial failure, environment differences, hidden coupling, resource saturation, and missing telemetry. Completion requires reproducible evidence, a causal explanation, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

- Define the objective, owner, authority, scope, assumptions, dependencies, and acceptance criteria for restart, rollback, and recovery.
- Model expected states, boundaries, inputs, outputs, failure modes, and evidence before implementation.
- Validate intended design, implemented behavior, runtime state, and user or mission outcome independently.
- Test positive, negative, boundary, degraded, failed, recovery, scale, and rollback conditions.
- Preserve source, version, configuration, data, telemetry, artifacts, timestamps, decisions, and limitations.
- Automate through typed contracts, least privilege, deterministic gates, bounded concurrency, canaries, and reversibility.

Failure patterns

- Treating restart, rollback, and recovery as a document or tool activity disconnected from actual system behavior.
- Relying on intended design or configuration without proving implementation and runtime outcomes.
- Ignoring failure, concurrency, data, identity, environment, compatibility, and operational boundaries.
- Changing several variables before preserving the original state and stating a falsifiable hypothesis.
- Allowing generated code, tests, findings, or optimizations to proceed without authoritative inputs and review.
- Declaring success after one positive example without negative tests, reproducibility, rollback, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, contracts, data, versions, ownership, and assumptions for restart, rollback, and recovery.
Observe	Runtime state, control flow, telemetry, artifacts, resource behavior, errors, test results, and user-visible outcomes.
Test	Expected, prohibited, boundary, malformed, concurrent, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, resource use, reliability, coverage, risk, maintainability, and operational impact.
Reconcile	Intended requirements and design against code, configuration, runtime behavior, tests, evidence, and residual limitations.

Chapter 06 laboratory and review

Practical laboratory
Design and test a service under dependency loss, overload, region loss, stale cache, and corrupted state, documenting each degradation contract.

Laboratory evidence package

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

Frontier extension

Explore self-healing architectures, local-first continuity, confidential recovery, and multi-region autonomous repair.

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

Observability, SLOs, and reliability evidence

Make failure behavior visible from code path to user outcome.

Chapter operating position
Make failure behavior visible from code path to user outcome.

7.1 Structured error telemetry

Structured error telemetry must be treated as an observable engineering mechanism rather than a slogan, framework checkbox, or tool output. Record error type, cause, operation, resource, dependency, retry, deadline, state, version, and correlation safely. The design should identify authoritative inputs, states, boundaries, assumptions, dependencies, limits, ownership, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the fault, detected condition, typed error, propagation, containment, recovery action, telemetry, and user outcome chain. A diagram, passing test, successful request, exploit result, benchmark, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with implemented state, negative and boundary tests, degraded conditions, raw evidence, recovery, and the resulting user, service, or risk outcome.

Implementation begins with a minimal known-good baseline and explicit invariants. Introduce one variable or dependency at a time, preserve before-state evidence, test expected and prohibited behavior, inject realistic failure, and retain a reversible path. Automation and AI assistance must stop safely when authority, freshness, identity, state, or expected outcomes are ambiguous.

Troubleshooting and assurance should locate the first divergence from the expected contract or state machine. Account for time, concurrency, caching, eventual consistency, data and schema drift, partial failure, environment differences, hidden coupling, resource saturation, and missing telemetry. Completion requires reproducible evidence, a causal explanation, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

- Define the objective, owner, authority, scope, assumptions, dependencies, and acceptance criteria for structured error telemetry.
- Model expected states, boundaries, inputs, outputs, failure modes, and evidence before implementation.
- Validate intended design, implemented behavior, runtime state, and user or mission outcome independently.
- Test positive, negative, boundary, degraded, failed, recovery, scale, and rollback conditions.
- Preserve source, version, configuration, data, telemetry, artifacts, timestamps, decisions, and limitations.
- Automate through typed contracts, least privilege, deterministic gates, bounded concurrency, canaries, and reversibility.

Failure patterns

- Treating structured error telemetry as a document or tool activity disconnected from actual system behavior.
- Relying on intended design or configuration without proving implementation and runtime outcomes.
- Ignoring failure, concurrency, data, identity, environment, compatibility, and operational boundaries.
- Changing several variables before preserving the original state and stating a falsifiable hypothesis.
- Allowing generated code, tests, findings, or optimizations to proceed without authoritative inputs and review.
- Declaring success after one positive example without negative tests, reproducibility, rollback, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, contracts, data, versions, ownership, and assumptions for structured error telemetry.
Observe	Runtime state, control flow, telemetry, artifacts, resource behavior, errors, test results, and user-visible outcomes.
Test	Expected, prohibited, boundary, malformed, concurrent, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, resource use, reliability, coverage, risk, maintainability, and operational impact.
Reconcile	Intended requirements and design against code, configuration, runtime behavior, tests, evidence, and residual limitations.

7.2 Traces, metrics, logs, and profiles

Traces, metrics, logs, and profiles must be treated as an observable engineering mechanism rather than a slogan, framework checkbox, or tool output. Connect request paths, queue time, dependency calls, saturation, exceptions, events, and resource consumption. The design should identify authoritative inputs, states, boundaries, assumptions, dependencies, limits, ownership, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the fault, detected condition, typed error, propagation, containment, recovery action, telemetry, and user outcome chain. A diagram, passing test, successful request, exploit result, benchmark, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with implemented state, negative and boundary tests, degraded conditions, raw evidence, recovery, and the resulting user, service, or risk outcome.

Implementation begins with a minimal known-good baseline and explicit invariants. Introduce one variable or dependency at a time, preserve before-state evidence, test expected and prohibited behavior, inject realistic failure, and retain a reversible path. Automation and AI assistance must stop safely when authority, freshness, identity, state, or expected outcomes are ambiguous.

Troubleshooting and assurance should locate the first divergence from the expected contract or state machine. Account for time, concurrency, caching, eventual consistency, data and schema drift, partial failure, environment differences, hidden coupling, resource saturation, and missing telemetry. Completion requires reproducible evidence, a causal explanation, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

- Define the objective, owner, authority, scope, assumptions, dependencies, and acceptance criteria for traces, metrics, logs, and profiles.
- Model expected states, boundaries, inputs, outputs, failure modes, and evidence before implementation.
- Validate intended design, implemented behavior, runtime state, and user or mission outcome independently.
- Test positive, negative, boundary, degraded, failed, recovery, scale, and rollback conditions.
- Preserve source, version, configuration, data, telemetry, artifacts, timestamps, decisions, and limitations.
- Automate through typed contracts, least privilege, deterministic gates, bounded concurrency, canaries, and reversibility.

Implementation sketch

```
error_contract:
  type: dependency-timeout
  retryable: true
  operation_outcome: unknown
  retry_policy:
    max_attempts: 2
    budget_from_parent_deadline: true
    jitter: full
  idempotency_key: required
  telemetry: [dependency, elapsed, deadline, attempt, effect_state]
  fallback: explicit-degraded-response
```

Failure patterns

- Treating traces, metrics, logs, and profiles as a document or tool activity disconnected from actual system behavior.
- Relying on intended design or configuration without proving implementation and runtime outcomes.
- Ignoring failure, concurrency, data, identity, environment, compatibility, and operational boundaries.

Changing several variables before preserving the original state and stating a falsifiable hypothesis.

Allowing generated code, tests, findings, or optimizations to proceed without authoritative inputs and review.

Declaring success after one positive example without negative tests, reproducibility, rollback, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, contracts, data, versions, ownership, and assumptions for traces, metrics, logs, and profiles.
Observe	Runtime state, control flow, telemetry, artifacts, resource behavior, errors, test results, and user-visible outcomes.
Test	Expected, prohibited, boundary, malformed, concurrent, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, resource use, reliability, coverage, risk, maintainability, and operational impact.
Reconcile	Intended requirements and design against code, configuration, runtime behavior, tests, evidence, and residual limitations.

7.3 SLIs, SLOs, and error budgets

SLIs, SLOs, and error budgets must be treated as an observable engineering mechanism rather than a slogan, framework checkbox, or tool output. Measure availability, correctness, latency, freshness, durability, completion, and user-visible failure against objectives. The design should identify authoritative inputs, states, boundaries, assumptions, dependencies, limits, ownership, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the fault, detected condition, typed error, propagation, containment, recovery action, telemetry, and user outcome chain. A diagram, passing test, successful request, exploit result, benchmark, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with implemented state, negative and boundary tests, degraded conditions, raw evidence, recovery, and the resulting user, service, or risk outcome.

Implementation begins with a minimal known-good baseline and explicit invariants. Introduce one variable or dependency at a time, preserve before-state evidence, test expected and prohibited behavior, inject realistic failure, and retain a reversible path. Automation and AI assistance must stop safely when authority, freshness, identity, state, or expected outcomes are ambiguous.

Troubleshooting and assurance should locate the first divergence from the expected contract or state machine. Account for time, concurrency, caching, eventual consistency, data and schema drift, partial failure, environment differences, hidden coupling, resource saturation, and missing telemetry. Completion requires reproducible evidence, a causal explanation, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

Define the objective, owner, authority, scope, assumptions, dependencies, and acceptance criteria for slis, slos, and error budgets.

Model expected states, boundaries, inputs, outputs, failure modes, and evidence before implementation.

Validate intended design, implemented behavior, runtime state, and user or mission outcome independently.

Test positive, negative, boundary, degraded, failed, recovery, scale, and rollback conditions.

Preserve source, version, configuration, data, telemetry, artifacts, timestamps, decisions, and limitations.

Automate through typed contracts, least privilege, deterministic gates, bounded concurrency, canaries, and reversibility.

Failure patterns

Treating slis, slos, and error budgets as a document or tool activity disconnected from actual system behavior.

Relying on intended design or configuration without proving implementation and runtime outcomes.

Ignoring failure, concurrency, data, identity, environment, compatibility, and operational boundaries.

Changing several variables before preserving the original state and stating a falsifiable hypothesis.

Allowing generated code, tests, findings, or optimizations to proceed without authoritative inputs and review.

Declaring success after one positive example without negative tests, reproducibility, rollback, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, contracts, data, versions, ownership, and assumptions for slis, slos, and error budgets.
Observe	Runtime state, control flow, telemetry, artifacts, resource behavior, errors, test results, and user-visible outcomes.
Test	Expected, prohibited, boundary, malformed, concurrent, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, resource use, reliability, coverage, risk, maintainability, and operational impact.
Reconcile	Intended requirements and design against code, configuration, runtime behavior, tests, evidence, and residual limitations.

7.4 Incident and postmortem learning

Incident and postmortem learning must be treated as an observable engineering mechanism rather than a slogan, framework checkbox, or tool output. Use timelines, causal graphs, contributing conditions, remediation, owners, proof, and recurrence monitoring. The design should identify authoritative inputs, states, boundaries, assumptions, dependencies, limits, ownership, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the fault, detected condition, typed error, propagation, containment, recovery action, telemetry, and user outcome chain. A diagram, passing test, successful request, exploit result, benchmark, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with implemented state, negative and boundary tests, degraded conditions, raw evidence, recovery, and the resulting user, service, or risk outcome.

Implementation begins with a minimal known-good baseline and explicit invariants. Introduce one variable or dependency at a time, preserve before-state evidence, test expected and prohibited behavior, inject realistic failure, and retain a reversible path. Automation and AI assistance must stop safely when authority, freshness, identity, state, or expected outcomes are ambiguous.

Troubleshooting and assurance should locate the first divergence from the expected contract or state machine. Account for time, concurrency, caching, eventual consistency, data and schema drift, partial failure, environment differences, hidden coupling, resource saturation, and missing telemetry. Completion requires reproducible evidence, a causal explanation, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

- Define the objective, owner, authority, scope, assumptions, dependencies, and acceptance criteria for incident and postmortem learning.
- Model expected states, boundaries, inputs, outputs, failure modes, and evidence before implementation.
- Validate intended design, implemented behavior, runtime state, and user or mission outcome independently.
- Test positive, negative, boundary, degraded, failed, recovery, scale, and rollback conditions.
- Preserve source, version, configuration, data, telemetry, artifacts, timestamps, decisions, and limitations.
- Automate through typed contracts, least privilege, deterministic gates, bounded concurrency, canaries, and reversibility.

Failure patterns

- Treating incident and postmortem learning as a document or tool activity disconnected from actual system behavior.
- Relying on intended design or configuration without proving implementation and runtime outcomes.
- Ignoring failure, concurrency, data, identity, environment, compatibility, and operational boundaries.
- Changing several variables before preserving the original state and stating a falsifiable hypothesis.
- Allowing generated code, tests, findings, or optimizations to proceed without authoritative inputs and review.
- Declaring success after one positive example without negative tests, reproducibility, rollback, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, contracts, data, versions, ownership, and assumptions for incident and postmortem learning.
Observe	Runtime state, control flow, telemetry, artifacts, resource behavior, errors, test results, and user-visible outcomes.
Test	Expected, prohibited, boundary, malformed, concurrent, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, resource use, reliability, coverage, risk, maintainability, and operational impact.
Reconcile	Intended requirements and design against code, configuration, runtime behavior, tests, evidence, and residual limitations.

Chapter 07 laboratory and review

Practical laboratory
Instrument a failure scenario with OpenTelemetry-style signals and build an SLO and error-budget analysis from the resulting 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 causal telemetry, automated incident reconstruction, and reliability agents that must cite raw evidence.

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

CHAPTER 08

Resilience testing, governance, and evolution

Continuously prove failure assumptions as systems and dependencies change.

Chapter operating position
Continuously prove failure assumptions as systems and dependencies change.

8.1 Fault injection and chaos experiments

Fault injection and chaos experiments must be treated as an observable engineering mechanism rather than a slogan, framework checkbox, or tool output. Define hypothesis, steady state, scope, safeguards, abort, injected condition, observation, recovery, and learning. The design should identify authoritative inputs, states, boundaries, assumptions, dependencies, limits, ownership, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the fault, detected condition, typed error, propagation, containment, recovery action, telemetry, and user outcome chain. A diagram, passing test, successful request, exploit result, benchmark, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with implemented state, negative and boundary tests, degraded conditions, raw evidence, recovery, and the resulting user, service, or risk outcome.

Implementation begins with a minimal known-good baseline and explicit invariants. Introduce one variable or dependency at a time, preserve before-state evidence, test expected and prohibited behavior, inject realistic failure, and retain a reversible path. Automation and AI assistance must stop safely when authority, freshness, identity, state, or expected outcomes are ambiguous.

Troubleshooting and assurance should locate the first divergence from the expected contract or state machine. Account for time, concurrency, caching, eventual consistency, data and schema drift, partial failure, environment differences, hidden coupling, resource saturation, and missing telemetry. Completion requires reproducible evidence, a causal explanation, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

- Define the objective, owner, authority, scope, assumptions, dependencies, and acceptance criteria for fault injection and chaos experiments.
- Model expected states, boundaries, inputs, outputs, failure modes, and evidence before implementation.
- Validate intended design, implemented behavior, runtime state, and user or mission outcome independently.
- Test positive, negative, boundary, degraded, failed, recovery, scale, and rollback conditions.
- Preserve source, version, configuration, data, telemetry, artifacts, timestamps, decisions, and limitations.
- Automate through typed contracts, least privilege, deterministic gates, bounded concurrency, canaries, and reversibility.

Failure patterns

- Treating fault injection and chaos experiments as a document or tool activity disconnected from actual system behavior.
- Relying on intended design or configuration without proving implementation and runtime outcomes.
- Ignoring failure, concurrency, data, identity, environment, compatibility, and operational boundaries.
- Changing several variables before preserving the original state and stating a falsifiable hypothesis.
- Allowing generated code, tests, findings, or optimizations to proceed without authoritative inputs and review.
- Declaring success after one positive example without negative tests, reproducibility, rollback, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, contracts, data, versions, ownership, and assumptions for fault injection and chaos experiments.
Observe	Runtime state, control flow, telemetry, artifacts, resource behavior, errors, test results, and user-visible outcomes.
Test	Expected, prohibited, boundary, malformed, concurrent, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, resource use, reliability, coverage, risk, maintainability, and operational impact.
Reconcile	Intended requirements and design against code, configuration, runtime behavior, tests, evidence, and residual limitations.

8.2 Property and model-based failure testing

Property and model-based failure testing must be treated as an observable engineering mechanism rather than a slogan, framework checkbox, or tool output. Generate state transitions, timing, concurrency, invalid inputs, dependency outcomes, and invariants. The design should identify authoritative inputs, states, boundaries, assumptions, dependencies, limits, ownership, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the fault, detected condition, typed error, propagation, containment, recovery action, telemetry, and user outcome chain. A diagram, passing test, successful request, exploit result, benchmark, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with implemented state, negative and boundary tests, degraded conditions, raw evidence, recovery, and the resulting user, service, or risk outcome.

Implementation begins with a minimal known-good baseline and explicit invariants. Introduce one variable or dependency at a time, preserve before-state evidence, test expected and prohibited behavior, inject realistic failure, and retain a reversible path. Automation and AI assistance must stop safely when authority, freshness, identity, state, or expected outcomes are ambiguous.

Troubleshooting and assurance should locate the first divergence from the expected contract or state machine. Account for time, concurrency, caching, eventual consistency, data and schema drift, partial failure, environment differences, hidden coupling, resource saturation, and missing telemetry. Completion requires reproducible evidence, a causal explanation, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

- Define the objective, owner, authority, scope, assumptions, dependencies, and acceptance criteria for property and model-based failure testing.
- Model expected states, boundaries, inputs, outputs, failure modes, and evidence before implementation.
- Validate intended design, implemented behavior, runtime state, and user or mission outcome independently.
- Test positive, negative, boundary, degraded, failed, recovery, scale, and rollback conditions.
- Preserve source, version, configuration, data, telemetry, artifacts, timestamps, decisions, and limitations.
- Automate through typed contracts, least privilege, deterministic gates, bounded concurrency, canaries, and reversibility.

Implementation sketch

```
error_contract:
  type: dependency-timeout
  retryable: true
  operation_outcome: unknown
  retry_policy:
    max_attempts: 2
    budget_from_parent_deadline: true
    jitter: full
  idempotency_key: required
  telemetry: [dependency, elapsed, deadline, attempt, effect_state]
  fallback: explicit-degraded-response
```

Failure patterns

- Treating property and model-based failure testing as a document or tool activity disconnected from actual system behavior.
- Relying on intended design or configuration without proving implementation and runtime outcomes.

Ignoring failure, concurrency, data, identity, environment, compatibility, and operational boundaries.

Changing several variables before preserving the original state and stating a falsifiable hypothesis.

Allowing generated code, tests, findings, or optimizations to proceed without authoritative inputs and review.

Declaring success after one positive example without negative tests, reproducibility, rollback, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, contracts, data, versions, ownership, and assumptions for property and model-based failure testing.
Observe	Runtime state, control flow, telemetry, artifacts, resource behavior, errors, test results, and user-visible outcomes.
Test	Expected, prohibited, boundary, malformed, concurrent, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, resource use, reliability, coverage, risk, maintainability, and operational impact.
Reconcile	Intended requirements and design against code, configuration, runtime behavior, tests, evidence, and residual limitations.

8.3 Reliability reviews and release gates

Reliability reviews and release gates must be treated as an observable engineering mechanism rather than a slogan, framework checkbox, or tool output. Require capacity, dependency, retry, rollback, migration, backup, disaster, runbook, and observability evidence. The design should identify authoritative inputs, states, boundaries, assumptions, dependencies, limits, ownership, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the fault, detected condition, typed error, propagation, containment, recovery action, telemetry, and user outcome chain. A diagram, passing test, successful request, exploit result, benchmark, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with implemented state, negative and boundary tests, degraded conditions, raw evidence, recovery, and the resulting user, service, or risk outcome.

Implementation begins with a minimal known-good baseline and explicit invariants. Introduce one variable or dependency at a time, preserve before-state evidence, test expected and prohibited behavior, inject realistic failure, and retain a reversible path. Automation and AI assistance must stop safely when authority, freshness, identity, state, or expected outcomes are ambiguous.

Troubleshooting and assurance should locate the first divergence from the expected contract or state machine. Account for time, concurrency, caching, eventual consistency, data and schema drift, partial failure, environment differences, hidden coupling, resource saturation, and missing telemetry. Completion requires reproducible evidence, a causal explanation, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

Define the objective, owner, authority, scope, assumptions, dependencies, and acceptance criteria for reliability reviews and release gates.

Model expected states, boundaries, inputs, outputs, failure modes, and evidence before implementation.

Validate intended design, implemented behavior, runtime state, and user or mission outcome independently.

Test positive, negative, boundary, degraded, failed, recovery, scale, and rollback conditions.

Preserve source, version, configuration, data, telemetry, artifacts, timestamps, decisions, and limitations.

Automate through typed contracts, least privilege, deterministic gates, bounded concurrency, canaries, and reversibility.

Failure patterns

Treating reliability reviews and release gates as a document or tool activity disconnected from actual system behavior.

Relying on intended design or configuration without proving implementation and runtime outcomes.

Ignoring failure, concurrency, data, identity, environment, compatibility, and operational boundaries.

Changing several variables before preserving the original state and stating a falsifiable hypothesis.

Allowing generated code, tests, findings, or optimizations to proceed without authoritative inputs and review.

Declaring success after one positive example without negative tests, reproducibility, rollback, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, contracts, data, versions, ownership, and assumptions for reliability reviews and release gates.
Observe	Runtime state, control flow, telemetry, artifacts, resource behavior, errors, test results, and user-visible outcomes.
Test	Expected, prohibited, boundary, malformed, concurrent, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, resource use, reliability, coverage, risk, maintainability, and operational impact.
Reconcile	Intended requirements and design against code, configuration, runtime behavior, tests, evidence, and residual limitations.

8.4 Reliability debt and modernization

Reliability debt and modernization must be treated as an observable engineering mechanism rather than a slogan, framework checkbox, or tool output. Track unsafe retries, unbounded queues, hidden dependencies, fragile state, missing telemetry, manual recovery, and unsupported components. The design should identify authoritative inputs, states, boundaries, assumptions, dependencies, limits, ownership, telemetry, failure behavior, and acceptance criteria.

This guide traces the subject through the fault, detected condition, typed error, propagation, containment, recovery action, telemetry, and user outcome chain. A diagram, passing test, successful request, exploit result, benchmark, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with implemented state, negative and boundary tests, degraded conditions, raw evidence, recovery, and the resulting user, service, or risk outcome.

Implementation begins with a minimal known-good baseline and explicit invariants. Introduce one variable or dependency at a time, preserve before-state evidence, test expected and prohibited behavior, inject realistic failure, and retain a reversible path. Automation and AI assistance must stop safely when authority, freshness, identity, state, or expected outcomes are ambiguous.

Troubleshooting and assurance should locate the first divergence from the expected contract or state machine. Account for time, concurrency, caching, eventual consistency, data and schema drift, partial failure, environment differences, hidden coupling, resource saturation, and missing telemetry. Completion requires reproducible evidence, a causal explanation, validated restoration, residual limitations, and prevention or monitoring actions.

Engineering practices

- Define the objective, owner, authority, scope, assumptions, dependencies, and acceptance criteria for reliability debt and modernization.
- Model expected states, boundaries, inputs, outputs, failure modes, and evidence before implementation.
- Validate intended design, implemented behavior, runtime state, and user or mission outcome independently.
- Test positive, negative, boundary, degraded, failed, recovery, scale, and rollback conditions.
- Preserve source, version, configuration, data, telemetry, artifacts, timestamps, decisions, and limitations.
- Automate through typed contracts, least privilege, deterministic gates, bounded concurrency, canaries, and reversibility.

Failure patterns

- Treating reliability debt and modernization as a document or tool activity disconnected from actual system behavior.
- Relying on intended design or configuration without proving implementation and runtime outcomes.
- Ignoring failure, concurrency, data, identity, environment, compatibility, and operational boundaries.
- Changing several variables before preserving the original state and stating a falsifiable hypothesis.
- Allowing generated code, tests, findings, or optimizations to proceed without authoritative inputs and review.
- Declaring success after one positive example without negative tests, reproducibility, rollback, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, contracts, data, versions, ownership, and assumptions for reliability debt and modernization.
Observe	Runtime state, control flow, telemetry, artifacts, resource behavior, errors, test results, and user-visible outcomes.
Test	Expected, prohibited, boundary, malformed, concurrent, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, resource use, reliability, coverage, risk, maintainability, and operational impact.
Reconcile	Intended requirements and design against code, configuration, runtime behavior, tests, evidence, and residual limitations.

Chapter 08 laboratory and review

Practical laboratory

Build a resilience test plan and execute controlled latency, error, overload, cancellation, and recovery experiments with automatic abort conditions.

Laboratory evidence package

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

Frontier extension

Develop digital-twin failure simulations and governed agents that generate experiments but cannot exceed signed safety boundaries.

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. ISO / IEC - ISO/IEC 25010:2023 - Product Quality Model
Role: Current product-quality characteristics for systems and software.
Verified: 2026-07-26
Official source: https://www.iso.org/standard/78176.html

02. IETF / RFC Editor - RFC 9457 - Problem Details for HTTP APIs
Role: Machine-readable HTTP API error representation.
Verified: 2026-07-26
Official source: https://www.rfc-editor.org/info/rfc9457

03. IETF / RFC Editor - RFC 9110 - HTTP Semantics
Role: HTTP method, status, representation, caching, and semantic requirements.
Verified: 2026-07-26
Official source: https://www.rfc-editor.org/info/rfc9110

04. OpenTelemetry - OpenTelemetry Semantic Conventions
Role: Cross-language telemetry meaning for traces, metrics, logs, and resources.
Verified: 2026-07-26
Official source: https://opentelemetry.io/docs/specs/semconv/

05. Google - Site Reliability Engineering - Handling Overload
Role: Overload, admission control, load shedding, and capacity behavior.
Verified: 2026-07-26
Official source: https://sre.google/sre-book/handling-overload/

06. Google - Site Reliability Engineering - Addressing Cascading Failures
Role: Failure amplification, retries, resource exhaustion, and recovery guidance.
Verified: 2026-07-26
Official source: https://sre.google/sre-book/addressing-cascading-failures/

07. NIST - SP 800-218 - Secure Software Development Framework 1.1
Role: Secure development practices integrated into software lifecycle and acceptance.
Verified: 2026-07-26
Official source: https://csrc.nist.gov/pubs/sp/800/218/final

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-CLD; MYTHOS-DAT; MYTHOS-SEC
Publication class	Field Guide / Canonical Living Overview Guide
Skill levels	L1 Foundational; L2 Practitioner; L3 Advanced; L4 Frontier
Tags	error architecture, reliability, failure semantics, timeouts, retries, idempotency, backpressure, circuit breaker, graceful degradation, SLO
Canonical route	/library/field-guides/mythos-fg-swe-0004/

Living publication

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