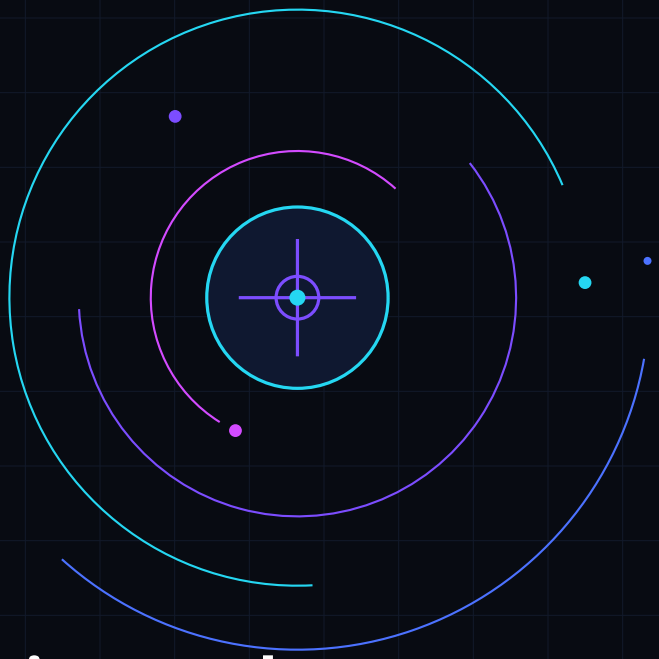

CANONICAL LIVING OVERVIEW GUIDE

Tauri, Svelte, TypeScript, and Secure Desktop Engineering

A secure desktop application guide covering Tauri 2 architecture, Rust core, webview threat model, commands and IPC, capabilities and permissions, CSP, plugins, file and shell access, Svelte 5 reactivity, TypeScript 6 contracts, state, accessibility, testing, packaging, signing, updates, observability, performance, multi-window and

PERMANENT ID
MYTHOS-FG-SWE-0019

PUBLICATION CLASS
FIELD GUIDE

VERSION / STATUS
1.0.0-candidate / CANDIDATE

PERMANENT PUBLICATION IDENTITY

Tauri, Svelte, TypeScript, and Secure Desktop Engineering

Document ID
MYTHOS-FG-SWE-0019

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-UX; MYTHOS-SEC; MYTHOS-CLD
Audience	Rust, Tauri, Svelte, TypeScript, desktop, product, UX, security, platform, and release engineers.
Prerequisites	Rust, JavaScript or TypeScript, web security, Svelte fundamentals, desktop operating systems, packaging, and testing.
Publisher / owner	Mythos Systems / Aaron Stovall
Public classification	Public technical guidance

Expected outcomes

Design a Tauri desktop application with a minimal trusted Rust core and least-privilege webview capabilities.

Create typed IPC contracts and validate every renderer-to-core request at the trust boundary.

Build Svelte 5 interfaces with explicit reactivity, accessible state, resilient workflows, and controlled side effects.

Package, sign, update, observe, test, and recover desktop applications across operating systems.

Profile startup, memory, IPC, rendering, I/O, and background work without weakening security.

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

Desktop architecture and threat model

Define the trusted computing base, process boundaries, content origins, data, and operating-system authority.

Chapter operating position
Define the trusted computing base, process boundaries, content origins, data, and operating-system authority.

1.1 Tauri process and webview model

Tauri process and webview model must be treated as an observable production mechanism rather than a framework slogan, code-style preference, or tool output. Separate Rust core, webview renderer, IPC, plugins, OS APIs, sidecars, storage, and remote services. The implementation should name authoritative inputs, types, states, ownership, security and concurrency boundaries, dependencies, limits, telemetry, failure behavior, cleanup, rollback, and acceptance criteria.

This guide traces the subject through the user intent, Svelte state, typed IPC, capability, Rust command, OS resource, result, update, telemetry, and recovery chain. A compiling build, passing unit test, successful request, valid type check, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with runtime state, negative and boundary tests, cancellation or partial failure, performance evidence, deployment identity, and the resulting user or service outcome.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one dependency, concurrency primitive, optimization, permission, or abstraction at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated code and AI assistance remain subject to the same contracts, review, tests, provenance, and release gates.

Troubleshooting and assurance locate the first divergence from the expected contract or state machine. Account for time, cancellation, ownership, aliasing, schema and version drift, caching, queues, resource saturation, hidden blocking, platform differences, partial deployment, and missing telemetry. Completion requires reproducibility, causal explanation, validated restoration, residual limitations, and prevention.

Engineering practices

- Define the objective, owner, authority, inputs, outputs, invariants, limits, and acceptance criteria for tauri process and webview model.
- Make types, lifecycle, trust boundaries, concurrency ownership, error states, and cleanup explicit.
- Validate design, source, build, runtime behavior, observability, and user outcome independently.
- Test positive, negative, malformed, boundary, cancellation, failure, recovery, scale, and rollback cases.
- Retain source, toolchain, dependency, artifact, configuration, telemetry, profile, and decision evidence.
- Prefer bounded work, least privilege, typed contracts, deterministic gates, staged rollout, and reversibility.

Failure patterns

- Treating tauri process and webview model as a library or syntax choice disconnected from end-to-end behavior.
- Assuming static types, framework defaults, or a successful build prove runtime correctness or security.
- Ignoring cancellation, cleanup, concurrency, version, environment, resource, or platform boundaries.
- Changing multiple variables before preserving the original state and stating a falsifiable hypothesis.
- Accepting generated code, dependencies, or optimizations without tests, provenance, review, and rollback.
- Declaring completion after one happy path without negative tests, profiling, deployment proof, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, types, contracts, dependencies, versions, ownership, and assumptions for tauri process and webview model.
Observe	Runtime state, tasks or goroutines, telemetry, artifacts, resources, errors, permissions, and user-visible outcomes.
Test	Expected, prohibited, malformed, boundary, concurrent, cancelled, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, allocations, memory, concurrency, reliability, security, and operator effort.
Reconcile	Intended design against code, build outputs, deployment identity, runtime behavior, tests, profiles, and residual limitations.

1.2 Assets and attacker paths

Assets and attacker paths must be treated as an observable production mechanism rather than a framework slogan, code-style preference, or tool output. Protect credentials, local data, updates, command authority, files, clipboard, deep links, notifications, and privileged automation. The implementation should name authoritative inputs, types, states, ownership, security and concurrency boundaries, dependencies, limits, telemetry, failure behavior, cleanup, rollback, and acceptance criteria.

This guide traces the subject through the user intent, Svelte state, typed IPC, capability, Rust command, OS resource, result, update, telemetry, and recovery chain. A compiling build, passing unit test, successful request, valid type check, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with runtime state, negative and boundary tests, cancellation or partial failure, performance evidence, deployment identity, and the resulting user or service outcome.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one dependency, concurrency primitive, optimization, permission, or abstraction at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated code and AI assistance remain subject to the same contracts, review, tests, provenance, and release gates.

Troubleshooting and assurance locate the first divergence from the expected contract or state machine. Account for time, cancellation, ownership, aliasing, schema and version drift, caching, queues, resource saturation, hidden blocking, platform differences, partial deployment, and missing telemetry. Completion requires reproducibility, causal explanation, validated restoration, residual limitations, and prevention.

Engineering practices

- Define the objective, owner, authority, inputs, outputs, invariants, limits, and acceptance criteria for assets and attacker paths.
- Make types, lifecycle, trust boundaries, concurrency ownership, error states, and cleanup explicit.
- Validate design, source, build, runtime behavior, observability, and user outcome independently.
- Test positive, negative, malformed, boundary, cancellation, failure, recovery, scale, and rollback cases.
- Retain source, toolchain, dependency, artifact, configuration, telemetry, profile, and decision evidence.
- Prefer bounded work, least privilege, typed contracts, deterministic gates, staged rollout, and reversibility.

Implementation sketch

```
csp:
  default-src: [self]
  script-src: [self]
  connect-src: [self, https://api.example]
  object-src: [none]
  frame-src: [none]
  base-uri: [none]
  form-action: [none]
  unsafe-inline: prohibited
```

Failure patterns

- Treating assets and attacker paths as a library or syntax choice disconnected from end-to-end behavior.
- Assuming static types, framework defaults, or a successful build prove runtime correctness or security.
- Ignoring cancellation, cleanup, concurrency, version, environment, resource, or platform boundaries.

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

Accepting generated code, dependencies, or optimizations without tests, provenance, review, and rollback.

Declaring completion after one happy path without negative tests, profiling, deployment proof, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, types, contracts, dependencies, versions, ownership, and assumptions for assets and attacker paths.
Observe	Runtime state, tasks or goroutines, telemetry, artifacts, resources, errors, permissions, and user-visible outcomes.
Test	Expected, prohibited, malformed, boundary, concurrent, cancelled, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, allocations, memory, concurrency, reliability, security, and operator effort.
Reconcile	Intended design against code, build outputs, deployment identity, runtime behavior, tests, profiles, and residual limitations.

1.3 Trust boundaries and content origins

Trust boundaries and content origins must be treated as an observable production mechanism rather than a framework slogan, code-style preference, or tool output. Distinguish bundled content, local custom protocols, remote content, user files, rendered HTML, plugins, and external links. The implementation should name authoritative inputs, types, states, ownership, security and concurrency boundaries, dependencies, limits, telemetry, failure behavior, cleanup, rollback, and acceptance criteria.

This guide traces the subject through the user intent, Svelte state, typed IPC, capability, Rust command, OS resource, result, update, telemetry, and recovery chain. A compiling build, passing unit test, successful request, valid type check, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with runtime state, negative and boundary tests, cancellation or partial failure, performance evidence, deployment identity, and the resulting user or service outcome.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one dependency, concurrency primitive, optimization, permission, or abstraction at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated code and AI assistance remain subject to the same contracts, review, tests, provenance, and release gates.

Troubleshooting and assurance locate the first divergence from the expected contract or state machine. Account for time, cancellation, ownership, aliasing, schema and version drift, caching, queues, resource saturation, hidden blocking, platform differences, partial deployment, and missing telemetry. Completion requires reproducibility, causal explanation, validated restoration, residual limitations, and prevention.

Engineering practices

Define the objective, owner, authority, inputs, outputs, invariants, limits, and acceptance criteria for trust boundaries and content origins.

Make types, lifecycle, trust boundaries, concurrency ownership, error states, and cleanup explicit.

Validate design, source, build, runtime behavior, observability, and user outcome independently.

Test positive, negative, malformed, boundary, cancellation, failure, recovery, scale, and rollback cases.

Retain source, toolchain, dependency, artifact, configuration, telemetry, profile, and decision evidence.

Prefer bounded work, least privilege, typed contracts, deterministic gates, staged rollout, and reversibility.

Failure patterns

Treating trust boundaries and content origins as a library or syntax choice disconnected from end-to-end behavior.

Assuming static types, framework defaults, or a successful build prove runtime correctness or security.

Ignoring cancellation, cleanup, concurrency, version, environment, resource, or platform boundaries.

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

Accepting generated code, dependencies, or optimizations without tests, provenance, review, and rollback.

Declaring completion after one happy path without negative tests, profiling, deployment proof, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, types, contracts, dependencies, versions, ownership, and assumptions for trust boundaries and content origins.
Observe	Runtime state, tasks or goroutines, telemetry, artifacts, resources, errors, permissions, and user-visible outcomes.
Test	Expected, prohibited, malformed, boundary, concurrent, cancelled, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, allocations, memory, concurrency, reliability, security, and operator effort.
Reconcile	Intended design against code, build outputs, deployment identity, runtime behavior, tests, profiles, and residual limitations.

1.4 Security and privacy requirements

Security and privacy requirements must be treated as an observable production mechanism rather than a framework slogan, code-style preference, or tool output. Define capabilities, data minimization, offline behavior, telemetry, crash data, recovery, and user consent. The implementation should name authoritative inputs, types, states, ownership, security and concurrency boundaries, dependencies, limits, telemetry, failure behavior, cleanup, rollback, and acceptance criteria.

This guide traces the subject through the user intent, Svelte state, typed IPC, capability, Rust command, OS resource, result, update, telemetry, and recovery chain. A compiling build, passing unit test, successful request, valid type check, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with runtime state, negative and boundary tests, cancellation or partial failure, performance evidence, deployment identity, and the resulting user or service outcome.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one dependency, concurrency primitive, optimization, permission, or abstraction at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated code and AI assistance remain subject to the same contracts, review, tests, provenance, and release gates.

Troubleshooting and assurance locate the first divergence from the expected contract or state machine. Account for time, cancellation, ownership, aliasing, schema and version drift, caching, queues, resource saturation, hidden blocking, platform differences, partial deployment, and missing telemetry. Completion requires reproducibility, causal explanation, validated restoration, residual limitations, and prevention.

Engineering practices

- Define the objective, owner, authority, inputs, outputs, invariants, limits, and acceptance criteria for security and privacy requirements.
- Make types, lifecycle, trust boundaries, concurrency ownership, error states, and cleanup explicit.
- Validate design, source, build, runtime behavior, observability, and user outcome independently.
- Test positive, negative, malformed, boundary, cancellation, failure, recovery, scale, and rollback cases.
- Retain source, toolchain, dependency, artifact, configuration, telemetry, profile, and decision evidence.
- Prefer bounded work, least privilege, typed contracts, deterministic gates, staged rollout, and reversibility.

Failure patterns

- Treating security and privacy requirements as a library or syntax choice disconnected from end-to-end behavior.
- Assuming static types, framework defaults, or a successful build prove runtime correctness or security.
- Ignoring cancellation, cleanup, concurrency, version, environment, resource, or platform boundaries.
- Changing multiple variables before preserving the original state and stating a falsifiable hypothesis.
- Accepting generated code, dependencies, or optimizations without tests, provenance, review, and rollback.
- Declaring completion after one happy path without negative tests, profiling, deployment proof, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, types, contracts, dependencies, versions, ownership, and assumptions for security and privacy requirements.
Observe	Runtime state, tasks or goroutines, telemetry, artifacts, resources, errors, permissions, and user-visible outcomes.
Test	Expected, prohibited, malformed, boundary, concurrent, cancelled, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, allocations, memory, concurrency, reliability, security, and operator effort.
Reconcile	Intended design against code, build outputs, deployment identity, runtime behavior, tests, profiles, and residual limitations.

Chapter 01 laboratory and review

Practical laboratory
Threat-model a Tauri desktop application with local files, remote APIs, updater, deep links, and background automation; derive security requirements.

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 the model to embodied AI presence, voice, agent tools, spatial displays, and confidential local inference.

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

Rust core, commands, IPC, and typed contracts

Treat every message from the webview as untrusted input crossing a privileged boundary.

Chapter operating position
Treat every message from the webview as untrusted input crossing a privileged boundary.

2.1 Command and state architecture

Command and state architecture must be treated as an observable production mechanism rather than a framework slogan, code-style preference, or tool output. Keep commands thin, validate input, authorize capability, call domain services, return typed results, and avoid global mutable state. The implementation should name authoritative inputs, types, states, ownership, security and concurrency boundaries, dependencies, limits, telemetry, failure behavior, cleanup, rollback, and acceptance criteria.

This guide traces the subject through the user intent, Svelte state, typed IPC, capability, Rust command, OS resource, result, update, telemetry, and recovery chain. A compiling build, passing unit test, successful request, valid type check, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with runtime state, negative and boundary tests, cancellation or partial failure, performance evidence, deployment identity, and the resulting user or service outcome.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one dependency, concurrency primitive, optimization, permission, or abstraction at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated code and AI assistance remain subject to the same contracts, review, tests, provenance, and release gates.

Troubleshooting and assurance locate the first divergence from the expected contract or state machine. Account for time, cancellation, ownership, aliasing, schema and version drift, caching, queues, resource saturation, hidden blocking, platform differences, partial deployment, and missing telemetry. Completion requires reproducibility, causal explanation, validated restoration, residual limitations, and prevention.

Engineering practices

- Define the objective, owner, authority, inputs, outputs, invariants, limits, and acceptance criteria for command and state architecture.
- Make types, lifecycle, trust boundaries, concurrency ownership, error states, and cleanup explicit.
- Validate design, source, build, runtime behavior, observability, and user outcome independently.
- Test positive, negative, malformed, boundary, cancellation, failure, recovery, scale, and rollback cases.
- Retain source, toolchain, dependency, artifact, configuration, telemetry, profile, and decision evidence.
- Prefer bounded work, least privilege, typed contracts, deterministic gates, staged rollout, and reversibility.

Failure patterns

- Treating command and state architecture as a library or syntax choice disconnected from end-to-end behavior.
- Assuming static types, framework defaults, or a successful build prove runtime correctness or security.
- Ignoring cancellation, cleanup, concurrency, version, environment, resource, or platform boundaries.
- Changing multiple variables before preserving the original state and stating a falsifiable hypothesis.
- Accepting generated code, dependencies, or optimizations without tests, provenance, review, and rollback.
- Declaring completion after one happy path without negative tests, profiling, deployment proof, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, types, contracts, dependencies, versions, ownership, and assumptions for command and state architecture.
Observe	Runtime state, tasks or goroutines, telemetry, artifacts, resources, errors, permissions, and user-visible outcomes.
Test	Expected, prohibited, malformed, boundary, concurrent, cancelled, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, allocations, memory, concurrency, reliability, security, and operator effort.
Reconcile	Intended design against code, build outputs, deployment identity, runtime behavior, tests, profiles, and residual limitations.

2.2 Serialization and schema contracts

Serialization and schema contracts must be treated as an observable production mechanism rather than a framework slogan, code-style preference, or tool output. Define request and response types, versioning, discriminated errors, unknown fields, size limits, and compatibility. The implementation should name authoritative inputs, types, states, ownership, security and concurrency boundaries, dependencies, limits, telemetry, failure behavior, cleanup, rollback, and acceptance criteria.

This guide traces the subject through the user intent, Svelte state, typed IPC, capability, Rust command, OS resource, result, update, telemetry, and recovery chain. A compiling build, passing unit test, successful request, valid type check, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with runtime state, negative and boundary tests, cancellation or partial failure, performance evidence, deployment identity, and the resulting user or service outcome.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one dependency, concurrency primitive, optimization, permission, or abstraction at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated code and AI assistance remain subject to the same contracts, review, tests, provenance, and release gates.

Troubleshooting and assurance locate the first divergence from the expected contract or state machine. Account for time, cancellation, ownership, aliasing, schema and version drift, caching, queues, resource saturation, hidden blocking, platform differences, partial deployment, and missing telemetry. Completion requires reproducibility, causal explanation, validated restoration, residual limitations, and prevention.

Engineering practices

- Define the objective, owner, authority, inputs, outputs, invariants, limits, and acceptance criteria for serialization and schema contracts.
- Make types, lifecycle, trust boundaries, concurrency ownership, error states, and cleanup explicit.
- Validate design, source, build, runtime behavior, observability, and user outcome independently.
- Test positive, negative, malformed, boundary, cancellation, failure, recovery, scale, and rollback cases.
- Retain source, toolchain, dependency, artifact, configuration, telemetry, profile, and decision evidence.
- Prefer bounded work, least privilege, typed contracts, deterministic gates, staged rollout, and reversibility.

Implementation sketch

```
csp:
  default-src: [self]
  script-src: [self]
  connect-src: [self, https://api.example]
  object-src: [none]
  frame-src: [none]
  base-uri: [none]
  form-action: [none]
  unsafe-inline: prohibited
```

Failure patterns

- Treating serialization and schema contracts as a library or syntax choice disconnected from end-to-end behavior.
- Assuming static types, framework defaults, or a successful build prove runtime correctness or security.
- Ignoring cancellation, cleanup, concurrency, version, environment, resource, or platform boundaries.

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

Accepting generated code, dependencies, or optimizations without tests, provenance, review, and rollback.

Declaring completion after one happy path without negative tests, profiling, deployment proof, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, types, contracts, dependencies, versions, ownership, and assumptions for serialization and schema contracts.
Observe	Runtime state, tasks or goroutines, telemetry, artifacts, resources, errors, permissions, and user-visible outcomes.
Test	Expected, prohibited, malformed, boundary, concurrent, cancelled, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, allocations, memory, concurrency, reliability, security, and operator effort.
Reconcile	Intended design against code, build outputs, deployment identity, runtime behavior, tests, profiles, and residual limitations.

2.3 IPC authentication and authorization

IPC authentication and authorization must be treated as an observable production mechanism rather than a framework slogan, code-style preference, or tool output. Bind window or webview, origin, capability, user intent, application state, target resource, and operation. The implementation should name authoritative inputs, types, states, ownership, security and concurrency boundaries, dependencies, limits, telemetry, failure behavior, cleanup, rollback, and acceptance criteria.

This guide traces the subject through the user intent, Svelte state, typed IPC, capability, Rust command, OS resource, result, update, telemetry, and recovery chain. A compiling build, passing unit test, successful request, valid type check, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with runtime state, negative and boundary tests, cancellation or partial failure, performance evidence, deployment identity, and the resulting user or service outcome.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one dependency, concurrency primitive, optimization, permission, or abstraction at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated code and AI assistance remain subject to the same contracts, review, tests, provenance, and release gates.

Troubleshooting and assurance locate the first divergence from the expected contract or state machine. Account for time, cancellation, ownership, aliasing, schema and version drift, caching, queues, resource saturation, hidden blocking, platform differences, partial deployment, and missing telemetry. Completion requires reproducibility, causal explanation, validated restoration, residual limitations, and prevention.

Engineering practices

Define the objective, owner, authority, inputs, outputs, invariants, limits, and acceptance criteria for ipc authentication and authorization.

Make types, lifecycle, trust boundaries, concurrency ownership, error states, and cleanup explicit.

Validate design, source, build, runtime behavior, observability, and user outcome independently.

Test positive, negative, malformed, boundary, cancellation, failure, recovery, scale, and rollback cases.

Retain source, toolchain, dependency, artifact, configuration, telemetry, profile, and decision evidence.

Prefer bounded work, least privilege, typed contracts, deterministic gates, staged rollout, and reversibility.

Failure patterns

Treating ipc authentication and authorization as a library or syntax choice disconnected from end-to-end behavior.

Assuming static types, framework defaults, or a successful build prove runtime correctness or security.

Ignoring cancellation, cleanup, concurrency, version, environment, resource, or platform boundaries.

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

Accepting generated code, dependencies, or optimizations without tests, provenance, review, and rollback.

Declaring completion after one happy path without negative tests, profiling, deployment proof, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, types, contracts, dependencies, versions, ownership, and assumptions for ipc authentication and authorization.
Observe	Runtime state, tasks or goroutines, telemetry, artifacts, resources, errors, permissions, and user-visible outcomes.
Test	Expected, prohibited, malformed, boundary, concurrent, cancelled, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, allocations, memory, concurrency, reliability, security, and operator effort.
Reconcile	Intended design against code, build outputs, deployment identity, runtime behavior, tests, profiles, and residual limitations.

2.4 Events, channels, and streaming

Events, channels, and streaming must be treated as an observable production mechanism rather than a framework slogan, code-style preference, or tool output. Control subscriptions, fan-out, backpressure, lifecycle, cancellation, window closure, and sensitive payloads. The implementation should name authoritative inputs, types, states, ownership, security and concurrency boundaries, dependencies, limits, telemetry, failure behavior, cleanup, rollback, and acceptance criteria.

This guide traces the subject through the user intent, Svelte state, typed IPC, capability, Rust command, OS resource, result, update, telemetry, and recovery chain. A compiling build, passing unit test, successful request, valid type check, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with runtime state, negative and boundary tests, cancellation or partial failure, performance evidence, deployment identity, and the resulting user or service outcome.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one dependency, concurrency primitive, optimization, permission, or abstraction at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated code and AI assistance remain subject to the same contracts, review, tests, provenance, and release gates.

Troubleshooting and assurance locate the first divergence from the expected contract or state machine. Account for time, cancellation, ownership, aliasing, schema and version drift, caching, queues, resource saturation, hidden blocking, platform differences, partial deployment, and missing telemetry. Completion requires reproducibility, causal explanation, validated restoration, residual limitations, and prevention.

Engineering practices

- Define the objective, owner, authority, inputs, outputs, invariants, limits, and acceptance criteria for events, channels, and streaming.
- Make types, lifecycle, trust boundaries, concurrency ownership, error states, and cleanup explicit.
- Validate design, source, build, runtime behavior, observability, and user outcome independently.
- Test positive, negative, malformed, boundary, cancellation, failure, recovery, scale, and rollback cases.
- Retain source, toolchain, dependency, artifact, configuration, telemetry, profile, and decision evidence.
- Prefer bounded work, least privilege, typed contracts, deterministic gates, staged rollout, and reversibility.

Failure patterns

- Treating events, channels, and streaming as a library or syntax choice disconnected from end-to-end behavior.
- Assuming static types, framework defaults, or a successful build prove runtime correctness or security.
- Ignoring cancellation, cleanup, concurrency, version, environment, resource, or platform boundaries.
- Changing multiple variables before preserving the original state and stating a falsifiable hypothesis.
- Accepting generated code, dependencies, or optimizations without tests, provenance, review, and rollback.
- Declaring completion after one happy path without negative tests, profiling, deployment proof, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, types, contracts, dependencies, versions, ownership, and assumptions for events, channels, and streaming.
Observe	Runtime state, tasks or goroutines, telemetry, artifacts, resources, errors, permissions, and user-visible outcomes.
Test	Expected, prohibited, malformed, boundary, concurrent, cancelled, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, allocations, memory, concurrency, reliability, security, and operator effort.
Reconcile	Intended design against code, build outputs, deployment identity, runtime behavior, tests, profiles, and residual limitations.

Chapter 02 laboratory and review

Practical laboratory
Implement a typed command and event channel, then test malformed data, unauthorized window, stale request, oversized payload, cancellation, and error mapping.

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 generated Rust-TypeScript contracts, capability tokens, signed command plans, and verifiable IPC transcripts.

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

Capabilities, permissions, plugins, and CSP

Minimize renderer authority and prevent content compromise from becoming operating-system compromise.

Chapter operating position
Minimize renderer authority and prevent content compromise from becoming operating-system compromise.

3.1 Capability files and window scope

Capability files and window scope must be treated as an observable production mechanism rather than a framework slogan, code-style preference, or tool output. Grant only named permissions to intended windows or webviews, platforms, local or remote origins, and feature states. The implementation should name authoritative inputs, types, states, ownership, security and concurrency boundaries, dependencies, limits, telemetry, failure behavior, cleanup, rollback, and acceptance criteria.

This guide traces the subject through the user intent, Svelte state, typed IPC, capability, Rust command, OS resource, result, update, telemetry, and recovery chain. A compiling build, passing unit test, successful request, valid type check, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with runtime state, negative and boundary tests, cancellation or partial failure, performance evidence, deployment identity, and the resulting user or service outcome.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one dependency, concurrency primitive, optimization, permission, or abstraction at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated code and AI assistance remain subject to the same contracts, review, tests, provenance, and release gates.

Troubleshooting and assurance locate the first divergence from the expected contract or state machine. Account for time, cancellation, ownership, aliasing, schema and version drift, caching, queues, resource saturation, hidden blocking, platform differences, partial deployment, and missing telemetry. Completion requires reproducibility, causal explanation, validated restoration, residual limitations, and prevention.

Engineering practices

- Define the objective, owner, authority, inputs, outputs, invariants, limits, and acceptance criteria for capability files and window scope.
- Make types, lifecycle, trust boundaries, concurrency ownership, error states, and cleanup explicit.
- Validate design, source, build, runtime behavior, observability, and user outcome independently.
- Test positive, negative, malformed, boundary, cancellation, failure, recovery, scale, and rollback cases.
- Retain source, toolchain, dependency, artifact, configuration, telemetry, profile, and decision evidence.
- Prefer bounded work, least privilege, typed contracts, deterministic gates, staged rollout, and reversibility.

Failure patterns

- Treating capability files and window scope as a library or syntax choice disconnected from end-to-end behavior.
- Assuming static types, framework defaults, or a successful build prove runtime correctness or security.
- Ignoring cancellation, cleanup, concurrency, version, environment, resource, or platform boundaries.
- Changing multiple variables before preserving the original state and stating a falsifiable hypothesis.
- Accepting generated code, dependencies, or optimizations without tests, provenance, review, and rollback.
- Declaring completion after one happy path without negative tests, profiling, deployment proof, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, types, contracts, dependencies, versions, ownership, and assumptions for capability files and window scope.
Observe	Runtime state, tasks or goroutines, telemetry, artifacts, resources, errors, permissions, and user-visible outcomes.
Test	Expected, prohibited, malformed, boundary, concurrent, cancelled, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, allocations, memory, concurrency, reliability, security, and operator effort.
Reconcile	Intended design against code, build outputs, deployment identity, runtime behavior, tests, profiles, and residual limitations.

3.2 Plugin and command permissions

Plugin and command permissions must be treated as an observable production mechanism rather than a framework slogan, code-style preference, or tool output. Review plugin commands, allow and deny sets, scopes, paths, URLs, shell programs, notifications, and update authority. The implementation should name authoritative inputs, types, states, ownership, security and concurrency boundaries, dependencies, limits, telemetry, failure behavior, cleanup, rollback, and acceptance criteria.

This guide traces the subject through the user intent, Svelte state, typed IPC, capability, Rust command, OS resource, result, update, telemetry, and recovery chain. A compiling build, passing unit test, successful request, valid type check, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with runtime state, negative and boundary tests, cancellation or partial failure, performance evidence, deployment identity, and the resulting user or service outcome.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one dependency, concurrency primitive, optimization, permission, or abstraction at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated code and AI assistance remain subject to the same contracts, review, tests, provenance, and release gates.

Troubleshooting and assurance locate the first divergence from the expected contract or state machine. Account for time, cancellation, ownership, aliasing, schema and version drift, caching, queues, resource saturation, hidden blocking, platform differences, partial deployment, and missing telemetry. Completion requires reproducibility, causal explanation, validated restoration, residual limitations, and prevention.

Engineering practices

- Define the objective, owner, authority, inputs, outputs, invariants, limits, and acceptance criteria for plugin and command permissions.
- Make types, lifecycle, trust boundaries, concurrency ownership, error states, and cleanup explicit.
- Validate design, source, build, runtime behavior, observability, and user outcome independently.
- Test positive, negative, malformed, boundary, cancellation, failure, recovery, scale, and rollback cases.
- Retain source, toolchain, dependency, artifact, configuration, telemetry, profile, and decision evidence.
- Prefer bounded work, least privilege, typed contracts, deterministic gates, staged rollout, and reversibility.

Implementation sketch

```
csp:
  default-src: [self]
  script-src: [self]
  connect-src: [self, https://api.example]
  object-src: [none]
  frame-src: [none]
  base-uri: [none]
  form-action: [none]
  unsafe-inline: prohibited
```

Failure patterns

- Treating plugin and command permissions as a library or syntax choice disconnected from end-to-end behavior.
- Assuming static types, framework defaults, or a successful build prove runtime correctness or security.
- Ignoring cancellation, cleanup, concurrency, version, environment, resource, or platform boundaries.

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

Accepting generated code, dependencies, or optimizations without tests, provenance, review, and rollback.

Declaring completion after one happy path without negative tests, profiling, deployment proof, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, types, contracts, dependencies, versions, ownership, and assumptions for plugin and command permissions.
Observe	Runtime state, tasks or goroutines, telemetry, artifacts, resources, errors, permissions, and user-visible outcomes.
Test	Expected, prohibited, malformed, boundary, concurrent, cancelled, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, allocations, memory, concurrency, reliability, security, and operator effort.
Reconcile	Intended design against code, build outputs, deployment identity, runtime behavior, tests, profiles, and residual limitations.

3.3 Content Security Policy

Content Security Policy must be treated as an observable production mechanism rather than a framework slogan, code-style preference, or tool output. Use restrictive default, script, style, connect, image, frame, object, base, and trusted content policies without unsafe bypasses. The implementation should name authoritative inputs, types, states, ownership, security and concurrency boundaries, dependencies, limits, telemetry, failure behavior, cleanup, rollback, and acceptance criteria.

This guide traces the subject through the user intent, Svelte state, typed IPC, capability, Rust command, OS resource, result, update, telemetry, and recovery chain. A compiling build, passing unit test, successful request, valid type check, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with runtime state, negative and boundary tests, cancellation or partial failure, performance evidence, deployment identity, and the resulting user or service outcome.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one dependency, concurrency primitive, optimization, permission, or abstraction at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated code and AI assistance remain subject to the same contracts, review, tests, provenance, and release gates.

Troubleshooting and assurance locate the first divergence from the expected contract or state machine. Account for time, cancellation, ownership, aliasing, schema and version drift, caching, queues, resource saturation, hidden blocking, platform differences, partial deployment, and missing telemetry. Completion requires reproducibility, causal explanation, validated restoration, residual limitations, and prevention.

Engineering practices

Define the objective, owner, authority, inputs, outputs, invariants, limits, and acceptance criteria for content security policy.

Make types, lifecycle, trust boundaries, concurrency ownership, error states, and cleanup explicit.

Validate design, source, build, runtime behavior, observability, and user outcome independently.

Test positive, negative, malformed, boundary, cancellation, failure, recovery, scale, and rollback cases.

Retain source, toolchain, dependency, artifact, configuration, telemetry, profile, and decision evidence.

Prefer bounded work, least privilege, typed contracts, deterministic gates, staged rollout, and reversibility.

Failure patterns

Treating content security policy as a library or syntax choice disconnected from end-to-end behavior.

Assuming static types, framework defaults, or a successful build prove runtime correctness or security.

Ignoring cancellation, cleanup, concurrency, version, environment, resource, or platform boundaries.

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

Accepting generated code, dependencies, or optimizations without tests, provenance, review, and rollback.

Declaring completion after one happy path without negative tests, profiling, deployment proof, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, types, contracts, dependencies, versions, ownership, and assumptions for content security policy.
Observe	Runtime state, tasks or goroutines, telemetry, artifacts, resources, errors, permissions, and user-visible outcomes.
Test	Expected, prohibited, malformed, boundary, concurrent, cancelled, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, allocations, memory, concurrency, reliability, security, and operator effort.
Reconcile	Intended design against code, build outputs, deployment identity, runtime behavior, tests, profiles, and residual limitations.

3.4 Remote content and untrusted files

Remote content and untrusted files must be treated as an observable production mechanism rather than a framework slogan, code-style preference, or tool output. Avoid remote scripts, isolate previews, sanitize HTML, prevent custom-protocol abuse, and open external content safely. The implementation should name authoritative inputs, types, states, ownership, security and concurrency boundaries, dependencies, limits, telemetry, failure behavior, cleanup, rollback, and acceptance criteria.

This guide traces the subject through the user intent, Svelte state, typed IPC, capability, Rust command, OS resource, result, update, telemetry, and recovery chain. A compiling build, passing unit test, successful request, valid type check, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with runtime state, negative and boundary tests, cancellation or partial failure, performance evidence, deployment identity, and the resulting user or service outcome.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one dependency, concurrency primitive, optimization, permission, or abstraction at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated code and AI assistance remain subject to the same contracts, review, tests, provenance, and release gates.

Troubleshooting and assurance locate the first divergence from the expected contract or state machine. Account for time, cancellation, ownership, aliasing, schema and version drift, caching, queues, resource saturation, hidden blocking, platform differences, partial deployment, and missing telemetry. Completion requires reproducibility, causal explanation, validated restoration, residual limitations, and prevention.

Engineering practices

- Define the objective, owner, authority, inputs, outputs, invariants, limits, and acceptance criteria for remote content and untrusted files.
- Make types, lifecycle, trust boundaries, concurrency ownership, error states, and cleanup explicit.
- Validate design, source, build, runtime behavior, observability, and user outcome independently.
- Test positive, negative, malformed, boundary, cancellation, failure, recovery, scale, and rollback cases.
- Retain source, toolchain, dependency, artifact, configuration, telemetry, profile, and decision evidence.
- Prefer bounded work, least privilege, typed contracts, deterministic gates, staged rollout, and reversibility.

Failure patterns

- Treating remote content and untrusted files as a library or syntax choice disconnected from end-to-end behavior.
- Assuming static types, framework defaults, or a successful build prove runtime correctness or security.
- Ignoring cancellation, cleanup, concurrency, version, environment, resource, or platform boundaries.
- Changing multiple variables before preserving the original state and stating a falsifiable hypothesis.
- Accepting generated code, dependencies, or optimizations without tests, provenance, review, and rollback.
- Declaring completion after one happy path without negative tests, profiling, deployment proof, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, types, contracts, dependencies, versions, ownership, and assumptions for remote content and untrusted files.
Observe	Runtime state, tasks or goroutines, telemetry, artifacts, resources, errors, permissions, and user-visible outcomes.
Test	Expected, prohibited, malformed, boundary, concurrent, cancelled, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, allocations, memory, concurrency, reliability, security, and operator effort.
Reconcile	Intended design against code, build outputs, deployment identity, runtime behavior, tests, profiles, and residual limitations.

Chapter 03 laboratory and review

Practical laboratory
Create a least-privilege capability and CSP set, then attempt unauthorized file, shell, network, window, and injected-script operations.

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 dynamically issued capabilities, policy attestation, isolated webviews, and security-oriented plugin sandboxes.

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

Svelte 5 state, components, and operational UX

Build explicit, testable UI state that represents real workflow and backend authority.

Chapter operating position
Build explicit, testable UI state that represents real workflow and backend authority.

4.1 Runes and reactive state

Runes and reactive state must be treated as an observable production mechanism rather than a framework slogan, code-style preference, or tool output. Use state, derived values, effects, props, bindable values, and shared reactive modules with controlled dependencies. The implementation should name authoritative inputs, types, states, ownership, security and concurrency boundaries, dependencies, limits, telemetry, failure behavior, cleanup, rollback, and acceptance criteria.

This guide traces the subject through the user intent, Svelte state, typed IPC, capability, Rust command, OS resource, result, update, telemetry, and recovery chain. A compiling build, passing unit test, successful request, valid type check, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with runtime state, negative and boundary tests, cancellation or partial failure, performance evidence, deployment identity, and the resulting user or service outcome.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one dependency, concurrency primitive, optimization, permission, or abstraction at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated code and AI assistance remain subject to the same contracts, review, tests, provenance, and release gates.

Troubleshooting and assurance locate the first divergence from the expected contract or state machine. Account for time, cancellation, ownership, aliasing, schema and version drift, caching, queues, resource saturation, hidden blocking, platform differences, partial deployment, and missing telemetry. Completion requires reproducibility, causal explanation, validated restoration, residual limitations, and prevention.

Engineering practices

- Define the objective, owner, authority, inputs, outputs, invariants, limits, and acceptance criteria for runes and reactive state.
- Make types, lifecycle, trust boundaries, concurrency ownership, error states, and cleanup explicit.
- Validate design, source, build, runtime behavior, observability, and user outcome independently.
- Test positive, negative, malformed, boundary, cancellation, failure, recovery, scale, and rollback cases.
- Retain source, toolchain, dependency, artifact, configuration, telemetry, profile, and decision evidence.
- Prefer bounded work, least privilege, typed contracts, deterministic gates, staged rollout, and reversibility.

Failure patterns

- Treating runes and reactive state as a library or syntax choice disconnected from end-to-end behavior.
- Assuming static types, framework defaults, or a successful build prove runtime correctness or security.
- Ignoring cancellation, cleanup, concurrency, version, environment, resource, or platform boundaries.
- Changing multiple variables before preserving the original state and stating a falsifiable hypothesis.
- Accepting generated code, dependencies, or optimizations without tests, provenance, review, and rollback.
- Declaring completion after one happy path without negative tests, profiling, deployment proof, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, types, contracts, dependencies, versions, ownership, and assumptions for runes and reactive state.
Observe	Runtime state, tasks or goroutines, telemetry, artifacts, resources, errors, permissions, and user-visible outcomes.
Test	Expected, prohibited, malformed, boundary, concurrent, cancelled, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, allocations, memory, concurrency, reliability, security, and operator effort.
Reconcile	Intended design against code, build outputs, deployment identity, runtime behavior, tests, profiles, and residual limitations.

4.2 Component and feature boundaries

Component and feature boundaries must be treated as an observable production mechanism rather than a framework slogan, code-style preference, or tool output. Separate domain views, reusable controls, data access, IPC adapters, state machines, and design-system primitives. The implementation should name authoritative inputs, types, states, ownership, security and concurrency boundaries, dependencies, limits, telemetry, failure behavior, cleanup, rollback, and acceptance criteria.

This guide traces the subject through the user intent, Svelte state, typed IPC, capability, Rust command, OS resource, result, update, telemetry, and recovery chain. A compiling build, passing unit test, successful request, valid type check, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with runtime state, negative and boundary tests, cancellation or partial failure, performance evidence, deployment identity, and the resulting user or service outcome.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one dependency, concurrency primitive, optimization, permission, or abstraction at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated code and AI assistance remain subject to the same contracts, review, tests, provenance, and release gates.

Troubleshooting and assurance locate the first divergence from the expected contract or state machine. Account for time, cancellation, ownership, aliasing, schema and version drift, caching, queues, resource saturation, hidden blocking, platform differences, partial deployment, and missing telemetry. Completion requires reproducibility, causal explanation, validated restoration, residual limitations, and prevention.

Engineering practices

- Define the objective, owner, authority, inputs, outputs, invariants, limits, and acceptance criteria for component and feature boundaries.
- Make types, lifecycle, trust boundaries, concurrency ownership, error states, and cleanup explicit.
- Validate design, source, build, runtime behavior, observability, and user outcome independently.
- Test positive, negative, malformed, boundary, cancellation, failure, recovery, scale, and rollback cases.
- Retain source, toolchain, dependency, artifact, configuration, telemetry, profile, and decision evidence.
- Prefer bounded work, least privilege, typed contracts, deterministic gates, staged rollout, and reversibility.

Implementation sketch

```
csp:
  default-src: [self]
  script-src: [self]
  connect-src: [self, https://api.example]
  object-src: [none]
  frame-src: [none]
  base-uri: [none]
  form-action: [none]
  unsafe-inline: prohibited
```

Failure patterns

- Treating component and feature boundaries as a library or syntax choice disconnected from end-to-end behavior.
- Assuming static types, framework defaults, or a successful build prove runtime correctness or security.
- Ignoring cancellation, cleanup, concurrency, version, environment, resource, or platform boundaries.

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

Accepting generated code, dependencies, or optimizations without tests, provenance, review, and rollback.

Declaring completion after one happy path without negative tests, profiling, deployment proof, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, types, contracts, dependencies, versions, ownership, and assumptions for component and feature boundaries.
Observe	Runtime state, tasks or goroutines, telemetry, artifacts, resources, errors, permissions, and user-visible outcomes.
Test	Expected, prohibited, malformed, boundary, concurrent, cancelled, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, allocations, memory, concurrency, reliability, security, and operator effort.
Reconcile	Intended design against code, build outputs, deployment identity, runtime behavior, tests, profiles, and residual limitations.

4.3 Async UI and workflow state

Async UI and workflow state must be treated as an observable production mechanism rather than a framework slogan, code-style preference, or tool output. Represent idle, loading, partial, success, empty, denied, offline, degraded, cancelled, and recovery states explicitly. The implementation should name authoritative inputs, types, states, ownership, security and concurrency boundaries, dependencies, limits, telemetry, failure behavior, cleanup, rollback, and acceptance criteria.

This guide traces the subject through the user intent, Svelte state, typed IPC, capability, Rust command, OS resource, result, update, telemetry, and recovery chain. A compiling build, passing unit test, successful request, valid type check, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with runtime state, negative and boundary tests, cancellation or partial failure, performance evidence, deployment identity, and the resulting user or service outcome.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one dependency, concurrency primitive, optimization, permission, or abstraction at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated code and AI assistance remain subject to the same contracts, review, tests, provenance, and release gates.

Troubleshooting and assurance locate the first divergence from the expected contract or state machine. Account for time, cancellation, ownership, aliasing, schema and version drift, caching, queues, resource saturation, hidden blocking, platform differences, partial deployment, and missing telemetry. Completion requires reproducibility, causal explanation, validated restoration, residual limitations, and prevention.

Engineering practices

Define the objective, owner, authority, inputs, outputs, invariants, limits, and acceptance criteria for async ui and workflow state.

Make types, lifecycle, trust boundaries, concurrency ownership, error states, and cleanup explicit.

Validate design, source, build, runtime behavior, observability, and user outcome independently.

Test positive, negative, malformed, boundary, cancellation, failure, recovery, scale, and rollback cases.

Retain source, toolchain, dependency, artifact, configuration, telemetry, profile, and decision evidence.

Prefer bounded work, least privilege, typed contracts, deterministic gates, staged rollout, and reversibility.

Failure patterns

Treating async ui and workflow state as a library or syntax choice disconnected from end-to-end behavior.

Assuming static types, framework defaults, or a successful build prove runtime correctness or security.

Ignoring cancellation, cleanup, concurrency, version, environment, resource, or platform boundaries.

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

Accepting generated code, dependencies, or optimizations without tests, provenance, review, and rollback.

Declaring completion after one happy path without negative tests, profiling, deployment proof, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, types, contracts, dependencies, versions, ownership, and assumptions for async ui and workflow state.
Observe	Runtime state, tasks or goroutines, telemetry, artifacts, resources, errors, permissions, and user-visible outcomes.
Test	Expected, prohibited, malformed, boundary, concurrent, cancelled, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, allocations, memory, concurrency, reliability, security, and operator effort.
Reconcile	Intended design against code, build outputs, deployment identity, runtime behavior, tests, profiles, and residual limitations.

4.4 Accessibility and high-density operations

Accessibility and high-density operations must be treated as an observable production mechanism rather than a framework slogan, code-style preference, or tool output. Support keyboard, focus, screen readers, semantics, zoom, color-independent status, shortcuts, and multi-panel information. The implementation should name authoritative inputs, types, states, ownership, security and concurrency boundaries, dependencies, limits, telemetry, failure behavior, cleanup, rollback, and acceptance criteria.

This guide traces the subject through the user intent, Svelte state, typed IPC, capability, Rust command, OS resource, result, update, telemetry, and recovery chain. A compiling build, passing unit test, successful request, valid type check, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with runtime state, negative and boundary tests, cancellation or partial failure, performance evidence, deployment identity, and the resulting user or service outcome.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one dependency, concurrency primitive, optimization, permission, or abstraction at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated code and AI assistance remain subject to the same contracts, review, tests, provenance, and release gates.

Troubleshooting and assurance locate the first divergence from the expected contract or state machine. Account for time, cancellation, ownership, aliasing, schema and version drift, caching, queues, resource saturation, hidden blocking, platform differences, partial deployment, and missing telemetry. Completion requires reproducibility, causal explanation, validated restoration, residual limitations, and prevention.

Engineering practices

- Define the objective, owner, authority, inputs, outputs, invariants, limits, and acceptance criteria for accessibility and high-density operations.
- Make types, lifecycle, trust boundaries, concurrency ownership, error states, and cleanup explicit.
- Validate design, source, build, runtime behavior, observability, and user outcome independently.
- Test positive, negative, malformed, boundary, cancellation, failure, recovery, scale, and rollback cases.
- Retain source, toolchain, dependency, artifact, configuration, telemetry, profile, and decision evidence.
- Prefer bounded work, least privilege, typed contracts, deterministic gates, staged rollout, and reversibility.

Failure patterns

- Treating accessibility and high-density operations as a library or syntax choice disconnected from end-to-end behavior.
- Assuming static types, framework defaults, or a successful build prove runtime correctness or security.
- Ignoring cancellation, cleanup, concurrency, version, environment, resource, or platform boundaries.
- Changing multiple variables before preserving the original state and stating a falsifiable hypothesis.
- Accepting generated code, dependencies, or optimizations without tests, provenance, review, and rollback.
- Declaring completion after one happy path without negative tests, profiling, deployment proof, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, types, contracts, dependencies, versions, ownership, and assumptions for accessibility and high-density operations.
Observe	Runtime state, tasks or goroutines, telemetry, artifacts, resources, errors, permissions, and user-visible outcomes.
Test	Expected, prohibited, malformed, boundary, concurrent, cancelled, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, allocations, memory, concurrency, reliability, security, and operator effort.
Reconcile	Intended design against code, build outputs, deployment identity, runtime behavior, tests, profiles, and residual limitations.

Chapter 04 laboratory and review

Practical laboratory
Build a Svelte workflow panel with explicit state transitions, optimistic and confirmed actions, error recovery, keyboard access, and backend reconciliation.

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 semantic UI state, adaptive information density, multimodal presence, and spatial operational workspaces.

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

TypeScript contracts, safety, and frontend architecture

Use TypeScript to prevent ambiguity while validating all runtime data and browser boundaries.

Chapter operating position
Use TypeScript to prevent ambiguity while validating all runtime data and browser boundaries.

5.1 Strict project configuration

Strict project configuration must be treated as an observable production mechanism rather than a framework slogan, code-style preference, or tool output. Enable strictness, exact optional semantics, unchecked index awareness, module clarity, isolated builds, and controlled library types. The implementation should name authoritative inputs, types, states, ownership, security and concurrency boundaries, dependencies, limits, telemetry, failure behavior, cleanup, rollback, and acceptance criteria.

This guide traces the subject through the user intent, Svelte state, typed IPC, capability, Rust command, OS resource, result, update, telemetry, and recovery chain. A compiling build, passing unit test, successful request, valid type check, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with runtime state, negative and boundary tests, cancellation or partial failure, performance evidence, deployment identity, and the resulting user or service outcome.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one dependency, concurrency primitive, optimization, permission, or abstraction at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated code and AI assistance remain subject to the same contracts, review, tests, provenance, and release gates.

Troubleshooting and assurance locate the first divergence from the expected contract or state machine. Account for time, cancellation, ownership, aliasing, schema and version drift, caching, queues, resource saturation, hidden blocking, platform differences, partial deployment, and missing telemetry. Completion requires reproducibility, causal explanation, validated restoration, residual limitations, and prevention.

Engineering practices

- Define the objective, owner, authority, inputs, outputs, invariants, limits, and acceptance criteria for strict project configuration.
- Make types, lifecycle, trust boundaries, concurrency ownership, error states, and cleanup explicit.
- Validate design, source, build, runtime behavior, observability, and user outcome independently.
- Test positive, negative, malformed, boundary, cancellation, failure, recovery, scale, and rollback cases.
- Retain source, toolchain, dependency, artifact, configuration, telemetry, profile, and decision evidence.
- Prefer bounded work, least privilege, typed contracts, deterministic gates, staged rollout, and reversibility.

Failure patterns

- Treating strict project configuration as a library or syntax choice disconnected from end-to-end behavior.
- Assuming static types, framework defaults, or a successful build prove runtime correctness or security.
- Ignoring cancellation, cleanup, concurrency, version, environment, resource, or platform boundaries.
- Changing multiple variables before preserving the original state and stating a falsifiable hypothesis.
- Accepting generated code, dependencies, or optimizations without tests, provenance, review, and rollback.
- Declaring completion after one happy path without negative tests, profiling, deployment proof, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, types, contracts, dependencies, versions, ownership, and assumptions for strict project configuration.
Observe	Runtime state, tasks or goroutines, telemetry, artifacts, resources, errors, permissions, and user-visible outcomes.
Test	Expected, prohibited, malformed, boundary, concurrent, cancelled, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, allocations, memory, concurrency, reliability, security, and operator effort.
Reconcile	Intended design against code, build outputs, deployment identity, runtime behavior, tests, profiles, and residual limitations.

5.2 Domain and IPC types

Domain and IPC types must be treated as an observable production mechanism rather than a framework slogan, code-style preference, or tool output. Use discriminated unions, branded identifiers, readonly data, exhaustive matching, result types, and generated schemas. The implementation should name authoritative inputs, types, states, ownership, security and concurrency boundaries, dependencies, limits, telemetry, failure behavior, cleanup, rollback, and acceptance criteria.

This guide traces the subject through the user intent, Svelte state, typed IPC, capability, Rust command, OS resource, result, update, telemetry, and recovery chain. A compiling build, passing unit test, successful request, valid type check, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with runtime state, negative and boundary tests, cancellation or partial failure, performance evidence, deployment identity, and the resulting user or service outcome.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one dependency, concurrency primitive, optimization, permission, or abstraction at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated code and AI assistance remain subject to the same contracts, review, tests, provenance, and release gates.

Troubleshooting and assurance locate the first divergence from the expected contract or state machine. Account for time, cancellation, ownership, aliasing, schema and version drift, caching, queues, resource saturation, hidden blocking, platform differences, partial deployment, and missing telemetry. Completion requires reproducibility, causal explanation, validated restoration, residual limitations, and prevention.

Engineering practices

- Define the objective, owner, authority, inputs, outputs, invariants, limits, and acceptance criteria for domain and ipc types.
- Make types, lifecycle, trust boundaries, concurrency ownership, error states, and cleanup explicit.
- Validate design, source, build, runtime behavior, observability, and user outcome independently.
- Test positive, negative, malformed, boundary, cancellation, failure, recovery, scale, and rollback cases.
- Retain source, toolchain, dependency, artifact, configuration, telemetry, profile, and decision evidence.
- Prefer bounded work, least privilege, typed contracts, deterministic gates, staged rollout, and reversibility.

Implementation sketch

```
type CommandResult<T> =
  | { kind: 'ok'; value: T; revision: string }
  | { kind: 'denied'; reason: string }
  | { kind: 'conflict'; currentRevision: string }
  | { kind: 'failed'; code: string; retryable: boolean };
```

Failure patterns

- Treating domain and ipc types as a library or syntax choice disconnected from end-to-end behavior.
- Assuming static types, framework defaults, or a successful build prove runtime correctness or security.
- Ignoring cancellation, cleanup, concurrency, version, environment, resource, or platform boundaries.
- Changing multiple variables before preserving the original state and stating a falsifiable hypothesis.
- Accepting generated code, dependencies, or optimizations without tests, provenance, review, and rollback.

Declaring completion after one happy path without negative tests, profiling, deployment proof, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, types, contracts, dependencies, versions, ownership, and assumptions for domain and ipc types.
Observe	Runtime state, tasks or goroutines, telemetry, artifacts, resources, errors, permissions, and user-visible outcomes.
Test	Expected, prohibited, malformed, boundary, concurrent, cancelled, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, allocations, memory, concurrency, reliability, security, and operator effort.
Reconcile	Intended design against code, build outputs, deployment identity, runtime behavior, tests, profiles, and residual limitations.

5.3 Runtime validation and unsafe escape hatches

Runtime validation and unsafe escape hatches must be treated as an observable production mechanism rather than a framework slogan, code-style preference, or tool output. Validate unknown data, constrain any, audit assertions, isolate browser APIs, and avoid prototype or object-injection hazards. The implementation should name authoritative inputs, types, states, ownership, security and concurrency boundaries, dependencies, limits, telemetry, failure behavior, cleanup, rollback, and acceptance criteria.

This guide traces the subject through the user intent, Svelte state, typed IPC, capability, Rust command, OS resource, result, update, telemetry, and recovery chain. A compiling build, passing unit test, successful request, valid type check, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with runtime state, negative and boundary tests, cancellation or partial failure, performance evidence, deployment identity, and the resulting user or service outcome.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one dependency, concurrency primitive, optimization, permission, or abstraction at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated code and AI assistance remain subject to the same contracts, review, tests, provenance, and release gates.

Troubleshooting and assurance locate the first divergence from the expected contract or state machine. Account for time, cancellation, ownership, aliasing, schema and version drift, caching, queues, resource saturation, hidden blocking, platform differences, partial deployment, and missing telemetry. Completion requires reproducibility, causal explanation, validated restoration, residual limitations, and prevention.

Engineering practices

Define the objective, owner, authority, inputs, outputs, invariants, limits, and acceptance criteria for runtime validation and unsafe escape hatches.

Make types, lifecycle, trust boundaries, concurrency ownership, error states, and cleanup explicit.

Validate design, source, build, runtime behavior, observability, and user outcome independently.

Test positive, negative, malformed, boundary, cancellation, failure, recovery, scale, and rollback cases.

Retain source, toolchain, dependency, artifact, configuration, telemetry, profile, and decision evidence.

Prefer bounded work, least privilege, typed contracts, deterministic gates, staged rollout, and reversibility.

Failure patterns

Treating runtime validation and unsafe escape hatches as a library or syntax choice disconnected from end-to-end behavior.

Assuming static types, framework defaults, or a successful build prove runtime correctness or security.

Ignoring cancellation, cleanup, concurrency, version, environment, resource, or platform boundaries.

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

Accepting generated code, dependencies, or optimizations without tests, provenance, review, and rollback.

Declaring completion after one happy path without negative tests, profiling, deployment proof, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, types, contracts, dependencies, versions, ownership, and assumptions for runtime validation and unsafe escape hatches.
Observe	Runtime state, tasks or goroutines, telemetry, artifacts, resources, errors, permissions, and user-visible outcomes.
Test	Expected, prohibited, malformed, boundary, concurrent, cancelled, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, allocations, memory, concurrency, reliability, security, and operator effort.
Reconcile	Intended design against code, build outputs, deployment identity, runtime behavior, tests, profiles, and residual limitations.

5.4 TypeScript 6 migration and compatibility

TypeScript 6 migration and compatibility must be treated as an observable production mechanism rather than a framework slogan, code-style preference, or tool output. Manage deprecations, module resolution, build tools, generated declarations, frontend dependencies, and future native compiler transition. The implementation should name authoritative inputs, types, states, ownership, security and concurrency boundaries, dependencies, limits, telemetry, failure behavior, cleanup, rollback, and acceptance criteria.

This guide traces the subject through the user intent, Svelte state, typed IPC, capability, Rust command, OS resource, result, update, telemetry, and recovery chain. A compiling build, passing unit test, successful request, valid type check, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with runtime state, negative and boundary tests, cancellation or partial failure, performance evidence, deployment identity, and the resulting user or service outcome.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one dependency, concurrency primitive, optimization, permission, or abstraction at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated code and AI assistance remain subject to the same contracts, review, tests, provenance, and release gates.

Troubleshooting and assurance locate the first divergence from the expected contract or state machine. Account for time, cancellation, ownership, aliasing, schema and version drift, caching, queues, resource saturation, hidden blocking, platform differences, partial deployment, and missing telemetry. Completion requires reproducibility, causal explanation, validated restoration, residual limitations, and prevention.

Engineering practices

- Define the objective, owner, authority, inputs, outputs, invariants, limits, and acceptance criteria for typescript 6 migration and compatibility.
- Make types, lifecycle, trust boundaries, concurrency ownership, error states, and cleanup explicit.
- Validate design, source, build, runtime behavior, observability, and user outcome independently.
- Test positive, negative, malformed, boundary, cancellation, failure, recovery, scale, and rollback cases.
- Retain source, toolchain, dependency, artifact, configuration, telemetry, profile, and decision evidence.
- Prefer bounded work, least privilege, typed contracts, deterministic gates, staged rollout, and reversibility.

Failure patterns

- Treating typescript 6 migration and compatibility as a library or syntax choice disconnected from end-to-end behavior.
- Assuming static types, framework defaults, or a successful build prove runtime correctness or security.
- Ignoring cancellation, cleanup, concurrency, version, environment, resource, or platform boundaries.
- Changing multiple variables before preserving the original state and stating a falsifiable hypothesis.
- Accepting generated code, dependencies, or optimizations without tests, provenance, review, and rollback.
- Declaring completion after one happy path without negative tests, profiling, deployment proof, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, types, contracts, dependencies, versions, ownership, and assumptions for typescript 6 migration and compatibility.
Observe	Runtime state, tasks or goroutines, telemetry, artifacts, resources, errors, permissions, and user-visible outcomes.
Test	Expected, prohibited, malformed, boundary, concurrent, cancelled, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, allocations, memory, concurrency, reliability, security, and operator effort.
Reconcile	Intended design against code, build outputs, deployment identity, runtime behavior, tests, profiles, and residual limitations.

Chapter 05 laboratory and review

Practical laboratory
Create strict TypeScript contracts for commands and workflow state, then test runtime schema mismatch, unknown variants, stale clients, and unsafe assertions.

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 effect modeling, schema-derived exhaustive UI, and proof-carrying frontend contracts.

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

Local data, files, OS integration, and background work

Use desktop authority safely across filesystems, credentials, devices, processes, and operating-system services.

Chapter operating position

Use desktop authority safely across filesystems, credentials, devices, processes, and operating-system services.

6.1 Application data and settings

Application data and settings must be treated as an observable production mechanism rather than a framework slogan, code-style preference, or tool output. Choose config, data, cache, logs, database, migration, encryption, backup, reset, and multi-user behavior. The implementation should name authoritative inputs, types, states, ownership, security and concurrency boundaries, dependencies, limits, telemetry, failure behavior, cleanup, rollback, and acceptance criteria.

This guide traces the subject through the user intent, Svelte state, typed IPC, capability, Rust command, OS resource, result, update, telemetry, and recovery chain. A compiling build, passing unit test, successful request, valid type check, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with runtime state, negative and boundary tests, cancellation or partial failure, performance evidence, deployment identity, and the resulting user or service outcome.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one dependency, concurrency primitive, optimization, permission, or abstraction at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated code and AI assistance remain subject to the same contracts, review, tests, provenance, and release gates.

Troubleshooting and assurance locate the first divergence from the expected contract or state machine. Account for time, cancellation, ownership, aliasing, schema and version drift, caching, queues, resource saturation, hidden blocking, platform differences, partial deployment, and missing telemetry. Completion requires reproducibility, causal explanation, validated restoration, residual limitations, and prevention.

Engineering practices

- Define the objective, owner, authority, inputs, outputs, invariants, limits, and acceptance criteria for application data and settings.
- Make types, lifecycle, trust boundaries, concurrency ownership, error states, and cleanup explicit.
- Validate design, source, build, runtime behavior, observability, and user outcome independently.
- Test positive, negative, malformed, boundary, cancellation, failure, recovery, scale, and rollback cases.
- Retain source, toolchain, dependency, artifact, configuration, telemetry, profile, and decision evidence.
- Prefer bounded work, least privilege, typed contracts, deterministic gates, staged rollout, and reversibility.

Failure patterns

- Treating application data and settings as a library or syntax choice disconnected from end-to-end behavior.
- Assuming static types, framework defaults, or a successful build prove runtime correctness or security.
- Ignoring cancellation, cleanup, concurrency, version, environment, resource, or platform boundaries.
- Changing multiple variables before preserving the original state and stating a falsifiable hypothesis.
- Accepting generated code, dependencies, or optimizations without tests, provenance, review, and rollback.
- Declaring completion after one happy path without negative tests, profiling, deployment proof, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, types, contracts, dependencies, versions, ownership, and assumptions for application data and settings.
Observe	Runtime state, tasks or goroutines, telemetry, artifacts, resources, errors, permissions, and user-visible outcomes.
Test	Expected, prohibited, malformed, boundary, concurrent, cancelled, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, allocations, memory, concurrency, reliability, security, and operator effort.
Reconcile	Intended design against code, build outputs, deployment identity, runtime behavior, tests, profiles, and residual limitations.

6.2 Filesystem and user-selected files

Filesystem and user-selected files must be treated as an observable production mechanism rather than a framework slogan, code-style preference, or tool output. Use scoped access, canonical paths, symlink controls, size limits, atomic writes, temporary files, and explicit user intent. The implementation should name authoritative inputs, types, states, ownership, security and concurrency boundaries, dependencies, limits, telemetry, failure behavior, cleanup, rollback, and acceptance criteria.

This guide traces the subject through the user intent, Svelte state, typed IPC, capability, Rust command, OS resource, result, update, telemetry, and recovery chain. A compiling build, passing unit test, successful request, valid type check, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with runtime state, negative and boundary tests, cancellation or partial failure, performance evidence, deployment identity, and the resulting user or service outcome.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one dependency, concurrency primitive, optimization, permission, or abstraction at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated code and AI assistance remain subject to the same contracts, review, tests, provenance, and release gates.

Troubleshooting and assurance locate the first divergence from the expected contract or state machine. Account for time, cancellation, ownership, aliasing, schema and version drift, caching, queues, resource saturation, hidden blocking, platform differences, partial deployment, and missing telemetry. Completion requires reproducibility, causal explanation, validated restoration, residual limitations, and prevention.

Engineering practices

- Define the objective, owner, authority, inputs, outputs, invariants, limits, and acceptance criteria for filesystem and user-selected files.
- Make types, lifecycle, trust boundaries, concurrency ownership, error states, and cleanup explicit.
- Validate design, source, build, runtime behavior, observability, and user outcome independently.
- Test positive, negative, malformed, boundary, cancellation, failure, recovery, scale, and rollback cases.
- Retain source, toolchain, dependency, artifact, configuration, telemetry, profile, and decision evidence.
- Prefer bounded work, least privilege, typed contracts, deterministic gates, staged rollout, and reversibility.

Implementation sketch

```
file_import:
  selection: explicit-user-dialog
  scope: selected-file-only
  canonicalize: true
  max_bytes: 10485760
  parse_in: rust-core
  output_write: atomic-temp-rename
  audit: [path-hash, size, outcome]
```

Failure patterns

- Treating filesystem and user-selected files as a library or syntax choice disconnected from end-to-end behavior.
- Assuming static types, framework defaults, or a successful build prove runtime correctness or security.
- Ignoring cancellation, cleanup, concurrency, version, environment, resource, or platform boundaries.

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

Accepting generated code, dependencies, or optimizations without tests, provenance, review, and rollback.

Declaring completion after one happy path without negative tests, profiling, deployment proof, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, types, contracts, dependencies, versions, ownership, and assumptions for filesystem and user-selected files.
Observe	Runtime state, tasks or goroutines, telemetry, artifacts, resources, errors, permissions, and user-visible outcomes.
Test	Expected, prohibited, malformed, boundary, concurrent, cancelled, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, allocations, memory, concurrency, reliability, security, and operator effort.
Reconcile	Intended design against code, build outputs, deployment identity, runtime behavior, tests, profiles, and residual limitations.

6.3 Shell, sidecars, and native integration

Shell, sidecars, and native integration must be treated as an observable production mechanism rather than a framework slogan, code-style preference, or tool output. Prefer APIs over shell, allowlist binaries and arguments, isolate sidecars, control environment, output, timeout, and updates. The implementation should name authoritative inputs, types, states, ownership, security and concurrency boundaries, dependencies, limits, telemetry, failure behavior, cleanup, rollback, and acceptance criteria.

This guide traces the subject through the user intent, Svelte state, typed IPC, capability, Rust command, OS resource, result, update, telemetry, and recovery chain. A compiling build, passing unit test, successful request, valid type check, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with runtime state, negative and boundary tests, cancellation or partial failure, performance evidence, deployment identity, and the resulting user or service outcome.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one dependency, concurrency primitive, optimization, permission, or abstraction at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated code and AI assistance remain subject to the same contracts, review, tests, provenance, and release gates.

Troubleshooting and assurance locate the first divergence from the expected contract or state machine. Account for time, cancellation, ownership, aliasing, schema and version drift, caching, queues, resource saturation, hidden blocking, platform differences, partial deployment, and missing telemetry. Completion requires reproducibility, causal explanation, validated restoration, residual limitations, and prevention.

Engineering practices

Define the objective, owner, authority, inputs, outputs, invariants, limits, and acceptance criteria for shell, sidecars, and native integration.

Make types, lifecycle, trust boundaries, concurrency ownership, error states, and cleanup explicit.

Validate design, source, build, runtime behavior, observability, and user outcome independently.

Test positive, negative, malformed, boundary, cancellation, failure, recovery, scale, and rollback cases.

Retain source, toolchain, dependency, artifact, configuration, telemetry, profile, and decision evidence.

Prefer bounded work, least privilege, typed contracts, deterministic gates, staged rollout, and reversibility.

Failure patterns

Treating shell, sidecars, and native integration as a library or syntax choice disconnected from end-to-end behavior.

Assuming static types, framework defaults, or a successful build prove runtime correctness or security.

Ignoring cancellation, cleanup, concurrency, version, environment, resource, or platform boundaries.

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

Accepting generated code, dependencies, or optimizations without tests, provenance, review, and rollback.

Declaring completion after one happy path without negative tests, profiling, deployment proof, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, types, contracts, dependencies, versions, ownership, and assumptions for shell, sidecars, and native integration.
Observe	Runtime state, tasks or goroutines, telemetry, artifacts, resources, errors, permissions, and user-visible outcomes.
Test	Expected, prohibited, malformed, boundary, concurrent, cancelled, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, allocations, memory, concurrency, reliability, security, and operator effort.
Reconcile	Intended design against code, build outputs, deployment identity, runtime behavior, tests, profiles, and residual limitations.

6.4 Background tasks and lifecycle

Background tasks and lifecycle must be treated as an observable production mechanism rather than a framework slogan, code-style preference, or tool output. Own tasks in Rust, support cancellation, app suspend or resume, single instance, tray, window closure, and shutdown drain. The implementation should name authoritative inputs, types, states, ownership, security and concurrency boundaries, dependencies, limits, telemetry, failure behavior, cleanup, rollback, and acceptance criteria.

This guide traces the subject through the user intent, Svelte state, typed IPC, capability, Rust command, OS resource, result, update, telemetry, and recovery chain. A compiling build, passing unit test, successful request, valid type check, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with runtime state, negative and boundary tests, cancellation or partial failure, performance evidence, deployment identity, and the resulting user or service outcome.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one dependency, concurrency primitive, optimization, permission, or abstraction at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated code and AI assistance remain subject to the same contracts, review, tests, provenance, and release gates.

Troubleshooting and assurance locate the first divergence from the expected contract or state machine. Account for time, cancellation, ownership, aliasing, schema and version drift, caching, queues, resource saturation, hidden blocking, platform differences, partial deployment, and missing telemetry. Completion requires reproducibility, causal explanation, validated restoration, residual limitations, and prevention.

Engineering practices

Define the objective, owner, authority, inputs, outputs, invariants, limits, and acceptance criteria for background tasks and lifecycle.

Make types, lifecycle, trust boundaries, concurrency ownership, error states, and cleanup explicit.

Validate design, source, build, runtime behavior, observability, and user outcome independently.

Test positive, negative, malformed, boundary, cancellation, failure, recovery, scale, and rollback cases.

Retain source, toolchain, dependency, artifact, configuration, telemetry, profile, and decision evidence.

Prefer bounded work, least privilege, typed contracts, deterministic gates, staged rollout, and reversibility.

Failure patterns

Treating background tasks and lifecycle as a library or syntax choice disconnected from end-to-end behavior.

Assuming static types, framework defaults, or a successful build prove runtime correctness or security.

Ignoring cancellation, cleanup, concurrency, version, environment, resource, or platform boundaries.

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

Accepting generated code, dependencies, or optimizations without tests, provenance, review, and rollback.

Declaring completion after one happy path without negative tests, profiling, deployment proof, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, types, contracts, dependencies, versions, ownership, and assumptions for background tasks and lifecycle.
Observe	Runtime state, tasks or goroutines, telemetry, artifacts, resources, errors, permissions, and user-visible outcomes.
Test	Expected, prohibited, malformed, boundary, concurrent, cancelled, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, allocations, memory, concurrency, reliability, security, and operator effort.
Reconcile	Intended design against code, build outputs, deployment identity, runtime behavior, tests, profiles, and residual limitations.

Chapter 06 laboratory and review

Practical laboratory
Implement a file import and background processing workflow with scoped permissions, atomic output, cancellation, progress, crash recovery, and audit.

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 sandboxed sidecars, capability-based local automation, secure enclaves, and offline-first synchronization.

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

Testing, debugging, observability, and performance

Prove the complete desktop stack across Rust, IPC, webview, UI, OS, and release artifacts.

Chapter operating position
Prove the complete desktop stack across Rust, IPC, webview, UI, OS, and release artifacts.

7.1 Rust and contract testing

Rust and contract testing must be treated as an observable production mechanism rather than a framework slogan, code-style preference, or tool output. Test domain services, commands, permissions, serialization, migration, error mapping, cancellation, and platform adapters. The implementation should name authoritative inputs, types, states, ownership, security and concurrency boundaries, dependencies, limits, telemetry, failure behavior, cleanup, rollback, and acceptance criteria.

This guide traces the subject through the user intent, Svelte state, typed IPC, capability, Rust command, OS resource, result, update, telemetry, and recovery chain. A compiling build, passing unit test, successful request, valid type check, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with runtime state, negative and boundary tests, cancellation or partial failure, performance evidence, deployment identity, and the resulting user or service outcome.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one dependency, concurrency primitive, optimization, permission, or abstraction at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated code and AI assistance remain subject to the same contracts, review, tests, provenance, and release gates.

Troubleshooting and assurance locate the first divergence from the expected contract or state machine. Account for time, cancellation, ownership, aliasing, schema and version drift, caching, queues, resource saturation, hidden blocking, platform differences, partial deployment, and missing telemetry. Completion requires reproducibility, causal explanation, validated restoration, residual limitations, and prevention.

Engineering practices

- Define the objective, owner, authority, inputs, outputs, invariants, limits, and acceptance criteria for rust and contract testing.
- Make types, lifecycle, trust boundaries, concurrency ownership, error states, and cleanup explicit.
- Validate design, source, build, runtime behavior, observability, and user outcome independently.
- Test positive, negative, malformed, boundary, cancellation, failure, recovery, scale, and rollback cases.
- Retain source, toolchain, dependency, artifact, configuration, telemetry, profile, and decision evidence.
- Prefer bounded work, least privilege, typed contracts, deterministic gates, staged rollout, and reversibility.

Failure patterns

- Treating rust and contract testing as a library or syntax choice disconnected from end-to-end behavior.
- Assuming static types, framework defaults, or a successful build prove runtime correctness or security.
- Ignoring cancellation, cleanup, concurrency, version, environment, resource, or platform boundaries.
- Changing multiple variables before preserving the original state and stating a falsifiable hypothesis.
- Accepting generated code, dependencies, or optimizations without tests, provenance, review, and rollback.
- Declaring completion after one happy path without negative tests, profiling, deployment proof, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, types, contracts, dependencies, versions, ownership, and assumptions for rust and contract testing.
Observe	Runtime state, tasks or goroutines, telemetry, artifacts, resources, errors, permissions, and user-visible outcomes.
Test	Expected, prohibited, malformed, boundary, concurrent, cancelled, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, allocations, memory, concurrency, reliability, security, and operator effort.
Reconcile	Intended design against code, build outputs, deployment identity, runtime behavior, tests, profiles, and residual limitations.

7.2 Frontend and end-to-end testing

Frontend and end-to-end testing must be treated as an observable production mechanism rather than a framework slogan, code-style preference, or tool output. Test components, state machines, accessibility, keyboard, IPC mocks, windows, dialogs, updates, and platform behavior. The implementation should name authoritative inputs, types, states, ownership, security and concurrency boundaries, dependencies, limits, telemetry, failure behavior, cleanup, rollback, and acceptance criteria.

This guide traces the subject through the user intent, Svelte state, typed IPC, capability, Rust command, OS resource, result, update, telemetry, and recovery chain. A compiling build, passing unit test, successful request, valid type check, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with runtime state, negative and boundary tests, cancellation or partial failure, performance evidence, deployment identity, and the resulting user or service outcome.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one dependency, concurrency primitive, optimization, permission, or abstraction at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated code and AI assistance remain subject to the same contracts, review, tests, provenance, and release gates.

Troubleshooting and assurance locate the first divergence from the expected contract or state machine. Account for time, cancellation, ownership, aliasing, schema and version drift, caching, queues, resource saturation, hidden blocking, platform differences, partial deployment, and missing telemetry. Completion requires reproducibility, causal explanation, validated restoration, residual limitations, and prevention.

Engineering practices

- Define the objective, owner, authority, inputs, outputs, invariants, limits, and acceptance criteria for frontend and end-to-end testing.
- Make types, lifecycle, trust boundaries, concurrency ownership, error states, and cleanup explicit.
- Validate design, source, build, runtime behavior, observability, and user outcome independently.
- Test positive, negative, malformed, boundary, cancellation, failure, recovery, scale, and rollback cases.
- Retain source, toolchain, dependency, artifact, configuration, telemetry, profile, and decision evidence.
- Prefer bounded work, least privilege, typed contracts, deterministic gates, staged rollout, and reversibility.

Implementation sketch

```
test_matrix:
  rust: [unit, command, permission, migration]
  typescript: [contract, exhaustive-state]
  svelte: [component, accessibility, keyboard]
  desktop: [ipc, windows, dialog, update, shutdown]
  performance: [startup, memory, ipc-volume, render]
```

Failure patterns

- Treating frontend and end-to-end testing as a library or syntax choice disconnected from end-to-end behavior.
- Assuming static types, framework defaults, or a successful build prove runtime correctness or security.
- Ignoring cancellation, cleanup, concurrency, version, environment, resource, or platform boundaries.
- Changing multiple variables before preserving the original state and stating a falsifiable hypothesis.

Accepting generated code, dependencies, or optimizations without tests, provenance, review, and rollback.

Declaring completion after one happy path without negative tests, profiling, deployment proof, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, types, contracts, dependencies, versions, ownership, and assumptions for frontend and end-to-end testing.
Observe	Runtime state, tasks or goroutines, telemetry, artifacts, resources, errors, permissions, and user-visible outcomes.
Test	Expected, prohibited, malformed, boundary, concurrent, cancelled, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, allocations, memory, concurrency, reliability, security, and operator effort.
Reconcile	Intended design against code, build outputs, deployment identity, runtime behavior, tests, profiles, and residual limitations.

7.3 Diagnostics and crash evidence

Diagnostics and crash evidence must be treated as an observable production mechanism rather than a framework slogan, code-style preference, or tool output. Collect structured Rust and frontend logs, traces, version, OS, window, command, error, update, and crash context with privacy. The implementation should name authoritative inputs, types, states, ownership, security and concurrency boundaries, dependencies, limits, telemetry, failure behavior, cleanup, rollback, and acceptance criteria.

This guide traces the subject through the user intent, Svelte state, typed IPC, capability, Rust command, OS resource, result, update, telemetry, and recovery chain. A compiling build, passing unit test, successful request, valid type check, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with runtime state, negative and boundary tests, cancellation or partial failure, performance evidence, deployment identity, and the resulting user or service outcome.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one dependency, concurrency primitive, optimization, permission, or abstraction at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated code and AI assistance remain subject to the same contracts, review, tests, provenance, and release gates.

Troubleshooting and assurance locate the first divergence from the expected contract or state machine. Account for time, cancellation, ownership, aliasing, schema and version drift, caching, queues, resource saturation, hidden blocking, platform differences, partial deployment, and missing telemetry. Completion requires reproducibility, causal explanation, validated restoration, residual limitations, and prevention.

Engineering practices

Define the objective, owner, authority, inputs, outputs, invariants, limits, and acceptance criteria for diagnostics and crash evidence.

Make types, lifecycle, trust boundaries, concurrency ownership, error states, and cleanup explicit.

Validate design, source, build, runtime behavior, observability, and user outcome independently.

Test positive, negative, malformed, boundary, cancellation, failure, recovery, scale, and rollback cases.

Retain source, toolchain, dependency, artifact, configuration, telemetry, profile, and decision evidence.

Prefer bounded work, least privilege, typed contracts, deterministic gates, staged rollout, and reversibility.

Failure patterns

Treating diagnostics and crash evidence as a library or syntax choice disconnected from end-to-end behavior.

Assuming static types, framework defaults, or a successful build prove runtime correctness or security.

Ignoring cancellation, cleanup, concurrency, version, environment, resource, or platform boundaries.

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

Accepting generated code, dependencies, or optimizations without tests, provenance, review, and rollback.

Declaring completion after one happy path without negative tests, profiling, deployment proof, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, types, contracts, dependencies, versions, ownership, and assumptions for diagnostics and crash evidence.
Observe	Runtime state, tasks or goroutines, telemetry, artifacts, resources, errors, permissions, and user-visible outcomes.
Test	Expected, prohibited, malformed, boundary, concurrent, cancelled, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, allocations, memory, concurrency, reliability, security, and operator effort.
Reconcile	Intended design against code, build outputs, deployment identity, runtime behavior, tests, profiles, and residual limitations.

7.4 Startup, memory, rendering, and IPC profiling

Startup, memory, rendering, and IPC profiling must be treated as an observable production mechanism rather than a framework slogan, code-style preference, or tool output. Measure binary and bundle size, startup phases, webview memory, reactivity, DOM, IPC volume, I/O, and background CPU. The implementation should name authoritative inputs, types, states, ownership, security and concurrency boundaries, dependencies, limits, telemetry, failure behavior, cleanup, rollback, and acceptance criteria.

This guide traces the subject through the user intent, Svelte state, typed IPC, capability, Rust command, OS resource, result, update, telemetry, and recovery chain. A compiling build, passing unit test, successful request, valid type check, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with runtime state, negative and boundary tests, cancellation or partial failure, performance evidence, deployment identity, and the resulting user or service outcome.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one dependency, concurrency primitive, optimization, permission, or abstraction at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated code and AI assistance remain subject to the same contracts, review, tests, provenance, and release gates.

Troubleshooting and assurance locate the first divergence from the expected contract or state machine. Account for time, cancellation, ownership, aliasing, schema and version drift, caching, queues, resource saturation, hidden blocking, platform differences, partial deployment, and missing telemetry. Completion requires reproducibility, causal explanation, validated restoration, residual limitations, and prevention.

Engineering practices

- Define the objective, owner, authority, inputs, outputs, invariants, limits, and acceptance criteria for startup, memory, rendering, and ipc profiling.
- Make types, lifecycle, trust boundaries, concurrency ownership, error states, and cleanup explicit.
- Validate design, source, build, runtime behavior, observability, and user outcome independently.
- Test positive, negative, malformed, boundary, cancellation, failure, recovery, scale, and rollback cases.
- Retain source, toolchain, dependency, artifact, configuration, telemetry, profile, and decision evidence.
- Prefer bounded work, least privilege, typed contracts, deterministic gates, staged rollout, and reversibility.

Failure patterns

- Treating startup, memory, rendering, and ipc profiling as a library or syntax choice disconnected from end-to-end behavior.
- Assuming static types, framework defaults, or a successful build prove runtime correctness or security.
- Ignoring cancellation, cleanup, concurrency, version, environment, resource, or platform boundaries.
- Changing multiple variables before preserving the original state and stating a falsifiable hypothesis.
- Accepting generated code, dependencies, or optimizations without tests, provenance, review, and rollback.
- Declaring completion after one happy path without negative tests, profiling, deployment proof, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, types, contracts, dependencies, versions, ownership, and assumptions for startup, memory, rendering, and ipc profiling.
Observe	Runtime state, tasks or goroutines, telemetry, artifacts, resources, errors, permissions, and user-visible outcomes.
Test	Expected, prohibited, malformed, boundary, concurrent, cancelled, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, allocations, memory, concurrency, reliability, security, and operator effort.
Reconcile	Intended design against code, build outputs, deployment identity, runtime behavior, tests, profiles, and residual limitations.

Chapter 07 laboratory and review

Practical laboratory
Build a cross-layer test and profiling suite, then diagnose a slow startup, memory growth, IPC storm, frozen UI, and failed background cancellation.

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 continuous desktop profiling, visual regression agents, semantic UI traces, and self-diagnosing local applications.

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

Packaging, signing, updates, and production operations

Ship trustworthy desktop software that installs, updates, rolls back, and supports users safely.

Chapter operating position
Ship trustworthy desktop software that installs, updates, rolls back, and supports users safely.

8.1 Platform packaging and entitlements

Platform packaging and entitlements must be treated as an observable production mechanism rather than a framework slogan, code-style preference, or tool output. Build Windows, macOS, and Linux artifacts with architecture, permissions, sandbox, services, icons, file associations, and uninstall. The implementation should name authoritative inputs, types, states, ownership, security and concurrency boundaries, dependencies, limits, telemetry, failure behavior, cleanup, rollback, and acceptance criteria.

This guide traces the subject through the user intent, Svelte state, typed IPC, capability, Rust command, OS resource, result, update, telemetry, and recovery chain. A compiling build, passing unit test, successful request, valid type check, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with runtime state, negative and boundary tests, cancellation or partial failure, performance evidence, deployment identity, and the resulting user or service outcome.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one dependency, concurrency primitive, optimization, permission, or abstraction at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated code and AI assistance remain subject to the same contracts, review, tests, provenance, and release gates.

Troubleshooting and assurance locate the first divergence from the expected contract or state machine. Account for time, cancellation, ownership, aliasing, schema and version drift, caching, queues, resource saturation, hidden blocking, platform differences, partial deployment, and missing telemetry. Completion requires reproducibility, causal explanation, validated restoration, residual limitations, and prevention.

Engineering practices

- Define the objective, owner, authority, inputs, outputs, invariants, limits, and acceptance criteria for platform packaging and entitlements.
- Make types, lifecycle, trust boundaries, concurrency ownership, error states, and cleanup explicit.
- Validate design, source, build, runtime behavior, observability, and user outcome independently.
- Test positive, negative, malformed, boundary, cancellation, failure, recovery, scale, and rollback cases.
- Retain source, toolchain, dependency, artifact, configuration, telemetry, profile, and decision evidence.
- Prefer bounded work, least privilege, typed contracts, deterministic gates, staged rollout, and reversibility.

Failure patterns

- Treating platform packaging and entitlements as a library or syntax choice disconnected from end-to-end behavior.
- Assuming static types, framework defaults, or a successful build prove runtime correctness or security.
- Ignoring cancellation, cleanup, concurrency, version, environment, resource, or platform boundaries.
- Changing multiple variables before preserving the original state and stating a falsifiable hypothesis.
- Accepting generated code, dependencies, or optimizations without tests, provenance, review, and rollback.
- Declaring completion after one happy path without negative tests, profiling, deployment proof, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, types, contracts, dependencies, versions, ownership, and assumptions for platform packaging and entitlements.
Observe	Runtime state, tasks or goroutines, telemetry, artifacts, resources, errors, permissions, and user-visible outcomes.
Test	Expected, prohibited, malformed, boundary, concurrent, cancelled, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, allocations, memory, concurrency, reliability, security, and operator effort.
Reconcile	Intended design against code, build outputs, deployment identity, runtime behavior, tests, profiles, and residual limitations.

8.2 Code signing and notarization

Code signing and notarization must be treated as an observable production mechanism rather than a framework slogan, code-style preference, or tool output. Protect signing identities, timestamps, entitlements, manifests, CI, certificate renewal, verification, and incident response. The implementation should name authoritative inputs, types, states, ownership, security and concurrency boundaries, dependencies, limits, telemetry, failure behavior, cleanup, rollback, and acceptance criteria.

This guide traces the subject through the user intent, Svelte state, typed IPC, capability, Rust command, OS resource, result, update, telemetry, and recovery chain. A compiling build, passing unit test, successful request, valid type check, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with runtime state, negative and boundary tests, cancellation or partial failure, performance evidence, deployment identity, and the resulting user or service outcome.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one dependency, concurrency primitive, optimization, permission, or abstraction at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated code and AI assistance remain subject to the same contracts, review, tests, provenance, and release gates.

Troubleshooting and assurance locate the first divergence from the expected contract or state machine. Account for time, cancellation, ownership, aliasing, schema and version drift, caching, queues, resource saturation, hidden blocking, platform differences, partial deployment, and missing telemetry. Completion requires reproducibility, causal explanation, validated restoration, residual limitations, and prevention.

Engineering practices

- Define the objective, owner, authority, inputs, outputs, invariants, limits, and acceptance criteria for code signing and notarization.
- Make types, lifecycle, trust boundaries, concurrency ownership, error states, and cleanup explicit.
- Validate design, source, build, runtime behavior, observability, and user outcome independently.
- Test positive, negative, malformed, boundary, cancellation, failure, recovery, scale, and rollback cases.
- Retain source, toolchain, dependency, artifact, configuration, telemetry, profile, and decision evidence.
- Prefer bounded work, least privilege, typed contracts, deterministic gates, staged rollout, and reversibility.

Implementation sketch

```
desktop_release:
  artifact_digest: sha256:...
  platform_signatures: [windows, macos, linux-package]
  notarization: required-where-applicable
  updater_signature: required
  rollout: [internal, canary, stable]
  rollback: signed-previous-version
```

Failure patterns

- Treating code signing and notarization as a library or syntax choice disconnected from end-to-end behavior.
- Assuming static types, framework defaults, or a successful build prove runtime correctness or security.
- Ignoring cancellation, cleanup, concurrency, version, environment, resource, or platform boundaries.
- Changing multiple variables before preserving the original state and stating a falsifiable hypothesis.

Accepting generated code, dependencies, or optimizations without tests, provenance, review, and rollback.

Declaring completion after one happy path without negative tests, profiling, deployment proof, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, types, contracts, dependencies, versions, ownership, and assumptions for code signing and notarization.
Observe	Runtime state, tasks or goroutines, telemetry, artifacts, resources, errors, permissions, and user-visible outcomes.
Test	Expected, prohibited, malformed, boundary, concurrent, cancelled, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, allocations, memory, concurrency, reliability, security, and operator effort.
Reconcile	Intended design against code, build outputs, deployment identity, runtime behavior, tests, profiles, and residual limitations.

8.3 Updater security and rollout

Updater security and rollout must be treated as an observable production mechanism rather than a framework slogan, code-style preference, or tool output. Verify signed metadata and artifacts, use staged channels, compatibility checks, anti-rollback policy, offline updates, and recovery. The implementation should name authoritative inputs, types, states, ownership, security and concurrency boundaries, dependencies, limits, telemetry, failure behavior, cleanup, rollback, and acceptance criteria.

This guide traces the subject through the user intent, Svelte state, typed IPC, capability, Rust command, OS resource, result, update, telemetry, and recovery chain. A compiling build, passing unit test, successful request, valid type check, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with runtime state, negative and boundary tests, cancellation or partial failure, performance evidence, deployment identity, and the resulting user or service outcome.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one dependency, concurrency primitive, optimization, permission, or abstraction at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated code and AI assistance remain subject to the same contracts, review, tests, provenance, and release gates.

Troubleshooting and assurance locate the first divergence from the expected contract or state machine. Account for time, cancellation, ownership, aliasing, schema and version drift, caching, queues, resource saturation, hidden blocking, platform differences, partial deployment, and missing telemetry. Completion requires reproducibility, causal explanation, validated restoration, residual limitations, and prevention.

Engineering practices

Define the objective, owner, authority, inputs, outputs, invariants, limits, and acceptance criteria for updater security and rollout.

Make types, lifecycle, trust boundaries, concurrency ownership, error states, and cleanup explicit.

Validate design, source, build, runtime behavior, observability, and user outcome independently.

Test positive, negative, malformed, boundary, cancellation, failure, recovery, scale, and rollback cases.

Retain source, toolchain, dependency, artifact, configuration, telemetry, profile, and decision evidence.

Prefer bounded work, least privilege, typed contracts, deterministic gates, staged rollout, and reversibility.

Failure patterns

Treating updater security and rollout as a library or syntax choice disconnected from end-to-end behavior.

Assuming static types, framework defaults, or a successful build prove runtime correctness or security.

Ignoring cancellation, cleanup, concurrency, version, environment, resource, or platform boundaries.

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

Accepting generated code, dependencies, or optimizations without tests, provenance, review, and rollback.

Declaring completion after one happy path without negative tests, profiling, deployment proof, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, types, contracts, dependencies, versions, ownership, and assumptions for updater security and rollout.
Observe	Runtime state, tasks or goroutines, telemetry, artifacts, resources, errors, permissions, and user-visible outcomes.
Test	Expected, prohibited, malformed, boundary, concurrent, cancelled, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, allocations, memory, concurrency, reliability, security, and operator effort.
Reconcile	Intended design against code, build outputs, deployment identity, runtime behavior, tests, profiles, and residual limitations.

8.4 Operations, support, and incident response

Operations, support, and incident response must be treated as an observable production mechanism rather than a framework slogan, code-style preference, or tool output. Maintain release inventory, diagnostics, privacy controls, support bundles, vulnerability response, revocation, and clean reinstall. The implementation should name authoritative inputs, types, states, ownership, security and concurrency boundaries, dependencies, limits, telemetry, failure behavior, cleanup, rollback, and acceptance criteria.

This guide traces the subject through the user intent, Svelte state, typed IPC, capability, Rust command, OS resource, result, update, telemetry, and recovery chain. A compiling build, passing unit test, successful request, valid type check, or green dashboard is not sufficient proof by itself. Intended behavior must be reconciled with runtime state, negative and boundary tests, cancellation or partial failure, performance evidence, deployment identity, and the resulting user or service outcome.

Implementation begins with a small, inspectable baseline and explicit invariants. Introduce one dependency, concurrency primitive, optimization, permission, or abstraction at a time; preserve before-state evidence; test expected and prohibited behavior; and retain a reversible path. Generated code and AI assistance remain subject to the same contracts, review, tests, provenance, and release gates.

Troubleshooting and assurance locate the first divergence from the expected contract or state machine. Account for time, cancellation, ownership, aliasing, schema and version drift, caching, queues, resource saturation, hidden blocking, platform differences, partial deployment, and missing telemetry. Completion requires reproducibility, causal explanation, validated restoration, residual limitations, and prevention.

Engineering practices

- Define the objective, owner, authority, inputs, outputs, invariants, limits, and acceptance criteria for operations, support, and incident response.
- Make types, lifecycle, trust boundaries, concurrency ownership, error states, and cleanup explicit.
- Validate design, source, build, runtime behavior, observability, and user outcome independently.
- Test positive, negative, malformed, boundary, cancellation, failure, recovery, scale, and rollback cases.
- Retain source, toolchain, dependency, artifact, configuration, telemetry, profile, and decision evidence.
- Prefer bounded work, least privilege, typed contracts, deterministic gates, staged rollout, and reversibility.

Failure patterns

- Treating operations, support, and incident response as a library or syntax choice disconnected from end-to-end behavior.
- Assuming static types, framework defaults, or a successful build prove runtime correctness or security.
- Ignoring cancellation, cleanup, concurrency, version, environment, resource, or platform boundaries.
- Changing multiple variables before preserving the original state and stating a falsifiable hypothesis.
- Accepting generated code, dependencies, or optimizations without tests, provenance, review, and rollback.
- Declaring completion after one happy path without negative tests, profiling, deployment proof, and recurrence checks.

Validation and evidence

Method	Expected evidence
Examine	Requirements, architecture, source, types, contracts, dependencies, versions, ownership, and assumptions for operations, support, and incident response.
Observe	Runtime state, tasks or goroutines, telemetry, artifacts, resources, errors, permissions, and user-visible outcomes.
Test	Expected, prohibited, malformed, boundary, concurrent, cancelled, degraded, failed, recovery, and rollback cases.
Measure	Correctness, latency, throughput, allocations, memory, concurrency, reliability, security, and operator effort.
Reconcile	Intended design against code, build outputs, deployment identity, runtime behavior, tests, profiles, and residual limitations.

Chapter 08 laboratory and review

Practical laboratory

Package and release a reference application through signed test artifacts, staged update, rollback, corrupted update, expired signer, and clean recovery scenarios.

Laboratory evidence package

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

Frontier extension

Develop a sovereign desktop release fabric with reproducible builds, post-quantum signing readiness, and verified offline distribution.

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. Tauri Project - Tauri 2.0 Documentation
Role: Cross-platform desktop and mobile application architecture with a Rust core and web frontend.
Verified: 2026-07-26
Official source: https://v2.tauri.app/

02. Tauri Project - Tauri Architecture
Role: Process model, webview, Rust core, and message-passing architecture.
Verified: 2026-07-26
Official source: https://v2.tauri.app/concept/architecture/

03. Tauri Project - Tauri Security
Role: Security model, threat considerations, and secure application practices.
Verified: 2026-07-26
Official source: https://v2.tauri.app/security/

04. Tauri Project - Content Security Policy
Role: CSP enforcement and secure content-loading guidance.
Verified: 2026-07-26
Official source: https://v2.tauri.app/security/csp/

05. Tauri Project - Capabilities
Role: Capability and permission scoping for windows and webviews.
Verified: 2026-07-26
Official source: https://v2.tauri.app/security/capabilities/

06. Svelte Project - What are runes?
Role: Svelte 5 explicit compiler-controlled reactivity model.
Verified: 2026-07-26
Official source: https://svelte.dev/docs/svelte/what-are-runes

07. Svelte Project - Svelte Best Practices
Role: Current guidance for robust and performant Svelte applications.
Verified: 2026-07-26
Official source: https://svelte.dev/docs/svelte/best-practices

08. Svelte Project - What is New in Svelte: July 2026
Role: Current toolchain, SvelteKit, and CLI evolution as of publication.
Verified: 2026-07-26
Official source: https://svelte.dev/blog/whats-new-in-svelte-july-2026

09. Microsoft - TypeScript 6.0 Release Notes
Role: Current TypeScript transition release, compatibility, and deprecations.
Verified: 2026-07-26
Official source: https://www.typescriptlang.org/docs/handbook/release-notes/typescript-6-0.html

10. Microsoft - TypeScript Handbook
Role: Type-system, narrowing, generics, modules, and project configuration guidance.
Verified: 2026-07-26
Official source: https://www.typescriptlang.org/docs/handbook/intro.html

11. Rust Project - Rust 1.97.1 Release
Role: Current stable Rust release status as of the publication date.
Verified: 2026-07-26
Official source: https://blog.rust-lang.org/releases/latest/

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-UX; MYTHOS-SEC; MYTHOS-CLD
Publication class	Field Guide / Canonical Living Overview Guide
Skill levels	L1 Foundational; L2 Practitioner; L3 Advanced; L4 Frontier
Tags	Tauri 2, Svelte 5, TypeScript 6, desktop engineering, IPC, capabilities, CSP, secure updates, accessibility, performance
Canonical route	/library/field-guides/mythos-fg-swe-0019/

Living publication

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