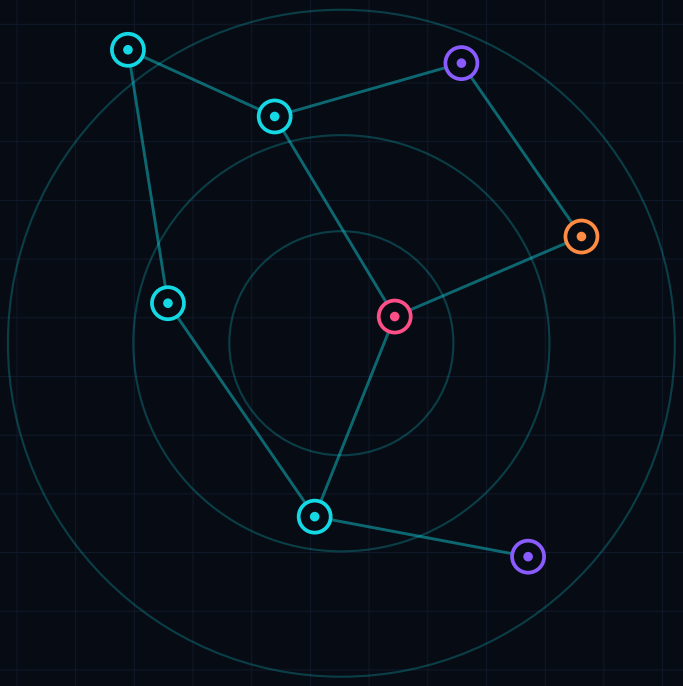
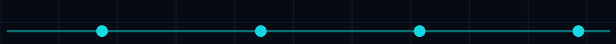
MYTHOS SYSTEMS

ENGINEERING STANDARDS LIBRARY / PORTFOLIO EDITION

Software Engineering Standards Portfolio

Architecture, languages, concurrency, typed boundaries, verification, interface protocols, debugging, and resilience.





PERMANENT DOCUMENT ID
MYTHOS-SWE-PORT-0001

PUBLICATION
Candidate Portfolio Edition

VERSION
1.0.0-candidate

PERMANENT PUBLICATION IDENTITY

Software Engineering Standards Portfolio

DOCUMENT ID
MYTHOS-SWE-PORT-0001

VERSION
1.0.0-candidate

STATUS
Candidate Portfolio Edition

PUBLISHER
Mythos Systems

PUBLICATION DATE
2026-09-17

CLASSIFICATION
Public

9	138	9	1
STANDARDS	CRITERIA	ATOMIC SOURCES	PERMANENT IDENTITY

Portfolio doctrine. This generated reader view does not replace its atomic source publications. Permanent IDs, evidence boundaries, assessment scope, freshness, and revision history remain authoritative at the atomic publication level.

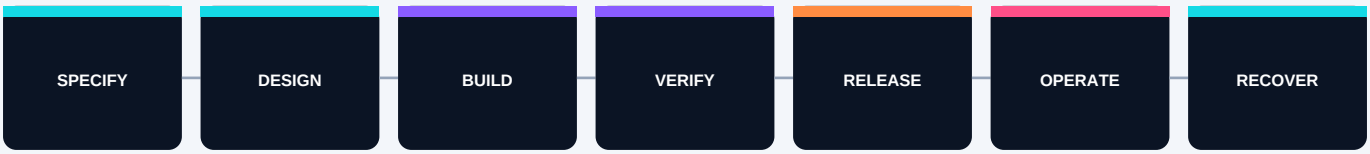
Publication domains

- Software architecture and domain design
- Rust, Go, and Python engineering
- Polyglot contracts and IPC
- Desktop and semantic UI architecture
- Formal verification, debugging, and resilience

PORTFOLIO READING MAP

From code to verified recovery

A visual navigation model for the software, platform, state, and reliability lifecycle.



STATE & DATA

contracts -> events -> durable state -> replicas -> storage -> evidence

SERVICE TOPOLOGY

API -> service mesh -> queues -> workers -> dependencies -> users

RELIABILITY

SLI -> SLO -> error budget -> capacity -> failure test -> recovery proof

ENGINEERING STANDARDS LIBRARY / SOFTWARE ENGINEERING

Advanced Software Design and Domain-Driven Design Standard

Domain boundaries, invariant-rich models, typed contracts, and architecture that keeps frameworks subordinate to business meaning.

PERMANENT DOCUMENT ID

MYTHOS-SWE-0001

PUBLICATION

Candidate Standard
Version 1.0.0-candidate

ASSURANCE PROFILE

Standard / Advanced
High Assurance

PERMANENT PUBLICATION IDENTITY

Advanced Software Design and Domain-Driven Design Standard

DOCUMENT ID
MYTHOS-SWE-0001

VERSION
1.0.0-candidate

STATUS
Candidate Standard

PUBLISHER
Mythos Systems

PUBLICATION DATE
2026-07-24

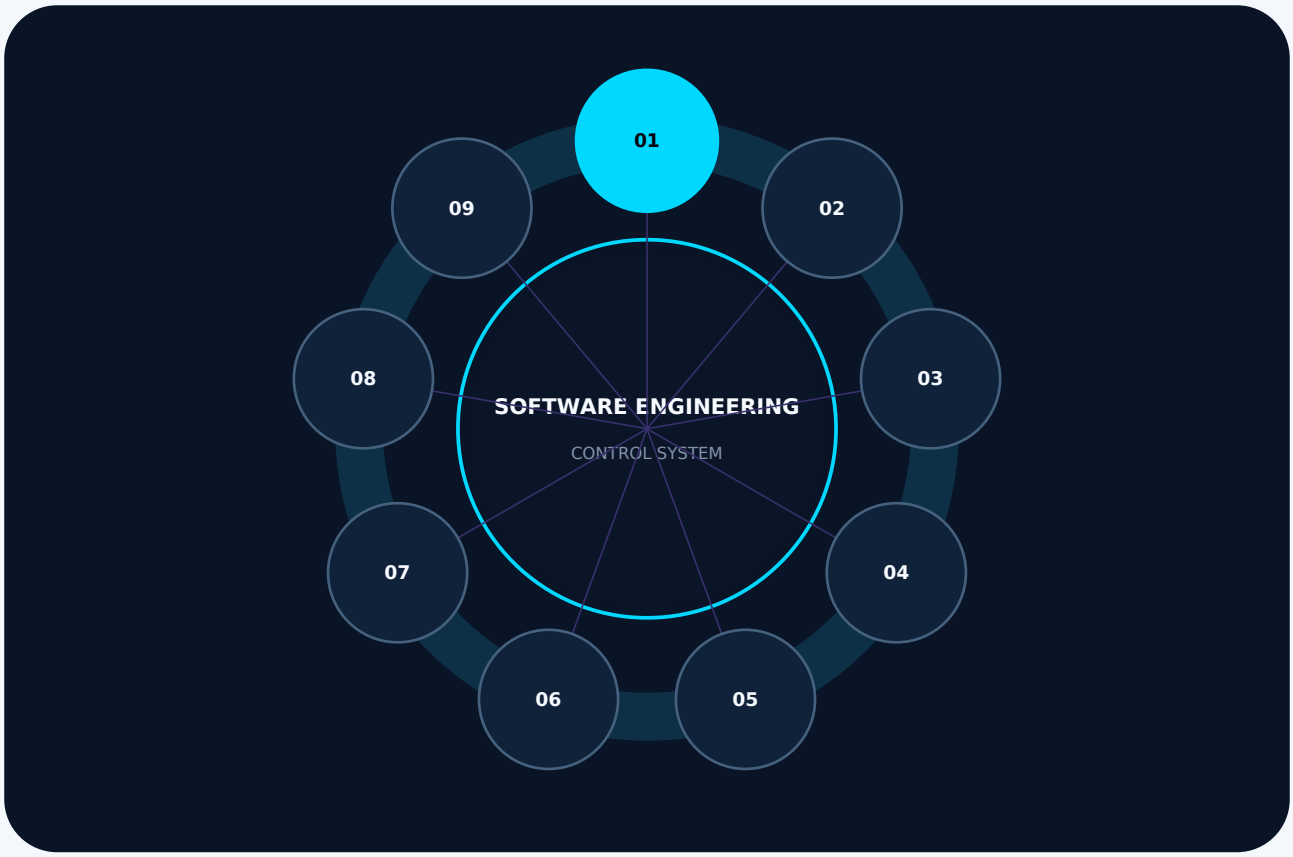
SCHEDULED REVIEW
2027-01-24

15	6	4	1
ATOMIC CRITERIA	ASSURANCE VIEWS	IMPLEMENTATION PROFILES	PERMANENT IDENTITY

Publication doctrine. This publication establishes durable guidance or requirements for its defined scope. Interpretation must preserve the permanent document identity, recorded evidence, controlled exceptions, and formal revision history.

DOMAIN CONTROL SYSTEM

MYTHOS-SWE-0001 inside the software engineering standard constellation



Connected assurance	Architecture, implementation, operations, evidence, and lifecycle are evaluated as one controlled system.
Specialization rule	Companion standards may add precision, but they cannot silently weaken this publication's requirements.

EXECUTIVE POSITION

Advanced Software Design and Domain-Driven Design Standard

The practice of software architecture has failed.

Evidence-bearing delivery path

Assurance maturity spectrum

CONTENTS

Publication navigation

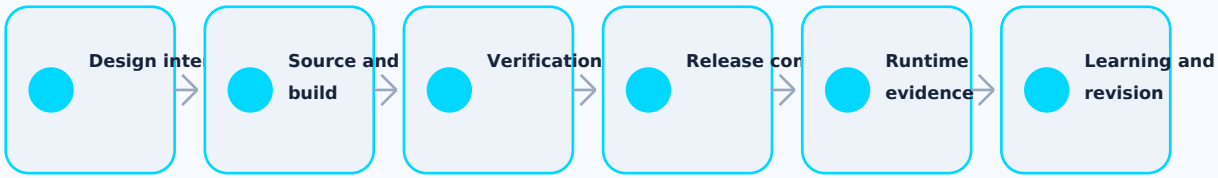
CONTROL TOPOLOGY

Assurance model and evaluation surface

The standard is assessed as an evidence-bearing system. Capability scores do not replace mandatory gates, implementation proof, or operational validation.

Capability family	Evaluation nodes
Bounded Contexts and Ubiquitous Language (Weight: 20%)	2
Domain Purity and Invariant Enforcement (Weight: 20%)	2
Architectural Boundaries (Hexagonal) (Weight: 20%)	2
Context Mapping and Anti-Corruption Layers (Weight: 15%)	1
Type Safety and Contracts at Boundaries (Weight: 15%)	1
Event-Driven and Temporal Design (Weight: 10%)	1
Suite Composition	2
Additional Evaluation Dimension	4

Lifecycle control loop



EVALUATION INDEX

Criteria and evidence map

#	Capability family	Criterion
1	Bounded Contexts and Ubiquitous Language (Weight: 20%)	Boundary Enforcement
2	Bounded Contexts and Ubiquitous Language (Weight: 20%)	Ubiquitous Language
3	Domain Purity and Invariant Enforcement (Weight: 20%)	Aggregate Invariants
4	Domain Purity and Invariant Enforcement (Weight: 20%)	Value Object Usage
5	Architectural Boundaries (Hexagonal) (Weight: 20%)	Dependency Rule
6	Architectural Boundaries (Hexagonal) (Weight: 20%)	Framework Independence
7	Context Mapping and Anti-Corruption Layers (Weight: 15%)	External Model Isolation
8	Type Safety and Contracts at Boundaries (Weight: 15%)	API Boundary Typing
9	Event-Driven and Temporal Design (Weight: 10%)	Domain Events
10	Suite Composition	MSDS-Purity
11	Suite Composition	MSDS-Boundaries
12	Additional Evaluation Dimension	Design Dimensions
13	Additional Evaluation Dimension	The Design Doctrine
14	Additional Evaluation Dimension	Scoring Model
15	Additional Evaluation Dimension	Suite Composition

NORMATIVE PUBLICATION

Requirements, evaluation methodology, and implementation guidance

The following body preserves the substantive source standard while normalizing publication status, identity, navigation, and visual presentation.

Executive Mandate

The practice of software architecture has failed.

Organizations build systems by reverse-engineering database schemas and wrapping them in framework controllers. They adopt microservices not because of domain boundaries, but because of organizational hype, resulting in distributed monoliths held together by synchronous REST calls and shared databases. They create "domain models" that are merely bags of getters and setters, stripping objects of behavior and scattering business logic across service layers.

This standard exists because:

1. Framework-Driven Design is Architectural Surrender: Designing your system around a web framework (Spring, Django, ASP.NET) or an ORM ensures your domain is subordinate to infrastructure. The domain MUST be the core; frameworks MUST be replaceable plugins.
2. The Anemic Domain Model is an Anti-Pattern: Objects that contain data but no behavior are not domain models; they are database transfer objects. Business logic MUST reside in the domain, enforced by strict aggregate invariants.
3. Bounded Contexts are Mandatory, Not Optional: A shared, ubiquitous "Customer" object across an entire enterprise is a fallacy. A customer in Billing is not a customer in Shipping. Systems MUST be decomposed into Bounded Contexts with explicit boundaries and translation maps.
4. Shared Databases are Integration Spaghetti: Integrating services via a shared relational database couples the structural internals of both systems, making independent deployment impossible. Integration MUST occur via well-defined APIs, asynchronous events, or Anti-Corruption Layers.

This document establishes the Mythos Software Design Standard (MSDS), a comprehensive, evidence-based methodology for evaluating software architectures against domain purity, boundary enforcement, and invariant correctness.

Scope and Applicability

In Scope

This standard covers the evaluation of:

1. Domain-Driven Design (DDD) strategic patterns (Bounded Contexts, Context Maps, Ubiquitous Language)
1. Domain-Driven Design tactical patterns (Aggregates, Entities, Value Objects, Domain Events)

1. Hexagonal Architecture (Ports and Adapters), Onion, and Clean Architecture
1. Anti-Corruption Layers (ACL) and integration patterns
1. Type-safe contracts and API boundary enforcement
1. Event-driven architecture and domain eventing
1. Invariant enforcement and type-state modeling

Out of Scope

1. Specific language engineering (covered in MYTHOS-SWE-0002 through 0004)
1. Formal verification state machines (covered in MYTHOS-SWE-0007)
1. General API security (covered in MYTHOS-SEC-0003)

Audience

1. Principal engineers and software architects
1. Domain modelers and business analysts
1. Platform engineering teams building internal frameworks
1. AI governance, risk, and compliance teams
1. Standards bodies seeking a reference software design methodology

Definitions and Taxonomy

Design Dimensions

Dimension	Definition
Bounded Context	A central pattern in DDD. A descriptive boundary within which a particular domain model and its Ubiquitous Language apply consistently.
Ubiquitous Language	A common, rigorous language used by both domain experts and developers to describe the domain model, ensuring zero translation errors.
Aggregate	A cluster of domain objects treated as a single unit for data changes. Enforces invariants and transactional consistency.
Hexagonal Architecture	An architectural pattern that isolates the domain logic from external concerns (UI, databases, APIs) by defining abstract "Ports" and concrete "Adapters".
Anti-Corruption Layer (ACL)	A translation layer that prevents the pollution of a clean domain model by the models of external or legacy systems.

The Design Doctrine

> The domain is the asset. The framework is a detail. The database is a detail. An anemic model is a bug. A shared database is an architectural failure. Behavior belongs where the data lives.

Evaluation Methodology

Scoring Model

Each capability is scored from 0 to 5.

Score	Meaning
0	Fails basic task; framework-driven, anemic models, shared DB
1	Basic DDD concepts but no strict boundaries
2	Functional but lacks domain purity or type enforcement
3	Solid production performance; hexagonal, aggregates, typed contracts
4	Strong performance; ACLs, event-driven, type-state modeling
5	Best-in-class; sets the standard for autonomous, formally verified domain design

Weighting

Category	Weight	Rationale
Bounded Contexts & Ubiquitous Language	20%	Without boundaries, you have a distributed monolith
Domain Purity & Invariant Enforcement	20%	Anemic models scatter logic; aggregates must protect invariants
Architectural Boundaries (Hexagonal)	20%	Framework lock-in kills agility
Context Mapping & Anti-Corruption Layers	15%	Unfiltered external models corrupt the domain
Type Safety & Contracts at Boundaries	15%	Stringly-typed APIs cause runtime failures
Event-Driven & Temporal Design	10%	Synchronous coupling limits resilience

Total: 100%

A system MUST score at least 3.0 in Bounded Contexts and Domain Purity to be considered for production use.

Evaluation Categories

Bounded Contexts and Ubiquitous Language (Weight: 20%)

Boundary Enforcement

Are bounded contexts explicitly defined and enforced, preventing model bleeding?

Score	Criteria
0	Single monolithic model shared across the entire organization
1	Logical separation in code but same database/deployment

Score	Criteria
2	Separate deployments but shared database or shared DTOs
3	Explicit bounded contexts with separate databases and independent lifecycles
4	Context maps defined (Partnership, Customer-Supplier, Conformist, ACL); integration via events/typed APIs
5	Perfect boundaries: formally verified context isolation, zero model bleeding, WASM-sandboxed integration, and automated conformance testing

Ubiquitous Language

Is the codebase aligned with the domain experts' language?

Score	Criteria
0	Technical jargon (e.g., `UserTable`, `DataManager`)
1	Partial alignment; some domain terms exist
2	Domain terms used in code but inconsistent with domain experts
3	Strict alignment: class/method names match domain language exactly
4	Language enforced via typed domain events and value objects
5	Perfect alignment: formally verified linguistic consistency, zero "translation" comments in code

Domain Purity and Invariant Enforcement (Weight: 20%)

Aggregate Invariants

Are business rules and invariants enforced inside aggregates, preventing invalid state?

Score	Criteria
0	Anemic domain model; logic in services, objects are just data
1	Basic validation in setters
2	Validation in service layer before persistence
3	Aggregates enforce invariants internally; no public setters
4	Type-state modeling (e.g., Rust enums/phantom types) makes invalid states unrepresentable
5	Perfect enforcement: formally verified invariants (TLA+/Apalache), zero invalid states possible, compile-time invariant checking

Value Object Usage

Are concepts defined by their attributes rather than identity modeled as Value Objects?

Score	Criteria
0	Primitive obsession (strings, ints used for domain concepts)
1	Some wrapper classes but no behavior
2	Value objects exist but are mutable
3	Immutable value objects with equality based on attributes
4	Value objects enforce domain rules (e.g., `EmailAddress` validates format at construction)

Score	Criteria
5	Perfect usage: rich, immutable value objects, zero primitive obsession, type-safe domain arithmetic

Architectural Boundaries (Hexagonal) (Weight: 20%)

Dependency Rule

Does the domain depend on infrastructure, or does infrastructure depend on the domain?

Score	Criteria
0	Domain imports framework/ORM annotations directly
1	Repository interfaces in domain, but implementations leak infrastructure concerns
2	Clean Hexagonal: domain has zero imports from infrastructure
3	Ports (interfaces) defined by domain; Adapters implement them
4	Domain is a pure, standalone module/framework-agnostic; infrastructure is a plugin
5	Perfect isolation: formally verified dependency graph, zero infrastructure coupling, domain runs without framework

Framework Independence

Can the framework (web, ORM, messaging) be replaced without modifying the domain?

Score	Criteria
0	Framework is the architecture; replacement requires rewrite
1	Framework abstractions exist but domain logic is scattered in controllers
2	Framework limited to infrastructure layer; domain unaware
3	Framework is a replaceable adapter; domain tests run without it
4	Multi-framework support proven (e.g., swapping Actix for Axum without domain changes)
5	Perfect independence: framework is a detail, domain is portable, formally verified isolation boundaries

Context Mapping and Anti-Corruption Layers (Weight: 15%)

External Model Isolation

Are external/legacy system models prevented from corrupting the core domain?

Score	Criteria
0	External models (e.g., Salesforce DTOs) used directly in domain logic
1	Translation logic scattered in service classes
2	Dedicated translation classes but no architectural boundary
3	Anti-Corruption Layer (ACL) implemented as a bounded context
4	ACL implemented as a stateful, event-driven translation gateway

Score	Criteria
5	Perfect isolation: WASM-sandboxed ACL, formally verified translation logic, zero external model leakage

Type Safety and Contracts at Boundaries (Weight: 15%)

API Boundary Typing

Are interactions between bounded contexts strictly typed?

Score	Criteria
0	Stringly-typed JSON / untyped dictionaries
1	Shared DTO libraries (coupling)
2	Interface definitions (OpenAPI/Protobuf) but unvalidated at runtime
3	Schema-validated, typed contracts (Protobuf/Smithy/gRPC) at all boundaries
4	Schema-validated + compiler-enforced types generated from contracts
5	Perfect contracts: formally verified schema compatibility, backward/forward compatibility guaranteed, automated contract testing

Event-Driven and Temporal Design (Weight: 10%)

Domain Events

Are state changes modeled as domain events?

Score	Criteria
0	CRUD only; no domain events
1	Events used for integration but not domain logic
2	Domain events trigger side effects within the same context
3	Event-sourced aggregates (state derived from events) for critical flows
4	CQRS (Command Query Responsibility Segregation) with separate read models
5	Perfect temporal design: event-sourced, formally verified event schemas, zero data loss, time-travel debugging

The Mythos Software Design Suite (MSDS)

Traditional software benchmarks measure code coverage. The MSDS evaluates domain purity, boundary isolation, and invariant correctness.

Suite Composition

MSDS-Purity

1. Anemic Model Detection: 50+ tests scanning for domain objects with public setters and no behavior.
1. Framework Leakage: 50+ tests checking domain module imports for framework dependencies.

MSDS-Boundaries

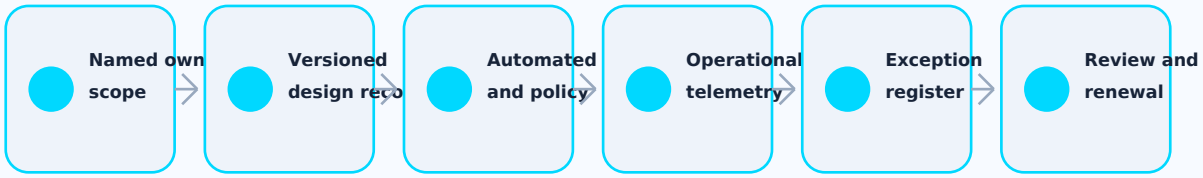
1. Invariant Violation: 100+ tests attempting to put aggregates into invalid states.
1. Contract Compatibility: 50+ tests verifying backward compatibility of API contracts across versions.

Execution Protocol

ASSURANCE AND ADOPTION

Implementation evidence and review gates

Required assurance chain



A passing score is not sufficient without current evidence. High-assurance adoption requires traceable design decisions, reproducible builds, independently reviewable tests, controlled secrets, runtime observability, failure exercises, and a dated exception register.

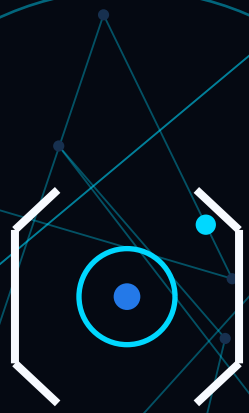
Assurance maturity spectrum

0	1	2	3	4	5
Absent	Ad hoc	Repeatable	Controlled	Measured	Adaptive

Current authority register

Authority	Version / snapshot	Applicability	Review by
Rust Project: The Rust Programming Language and Rust Reference	Rolling official documentation; book assumes Rust 1.90.0 or later and Rust 2024 Edition	High	2026-10-22
Go Project: The Go Programming Language Specification and Effective Go	Rolling official documentation snapshot	High	2026-10-22
Python Software Foundation: Python Language Reference, Packaging, Typing, and PEP Index	Rolling official documentation snapshot	High	2026-10-22
Tauri Programme: Tauri 2 Documentation	Tauri 2 rolling documentation snapshot	High	2026-10-22
Svelte Project: Svelte Documentation	Svelte 5 rolling documentation snapshot	High	2026-10-22
TLA+ Foundation: TLA+ Language and Tools Documentation	Rolling documentation snapshot	High	2027-01-24
OpenAPI Initiative: OpenAPI Specification	3.2.0	High	2027-01-24
OpenTelemetry Project: OpenTelemetry Documentation and Specification	Rolling specification snapshot; profiles public alpha and declarative configuration stable in 2026	High	2026-10-22

Research limitation. Official documentation establishes published interfaces and declared behavior. It does not prove the performance, security, interoperability, cost, or operational suitability of a specific implementation. Those claims require controlled testing and retained evidence.



ENGINEERING STANDARDS LIBRARY / SOFTWARE ENGINEERING

Rust Engineering, Memory Safety, and Concurrency Standard

Memory-safe systems engineering, explicit ownership, controlled unsafe code, concurrency correctness, and production Rust governance.

PERMANENT DOCUMENT ID

MYTHOS-SWE-0002

PUBLICATION

Candidate Standard
Version 1.0.0-candidate

ASSURANCE PROFILE

Standard / Advanced
High Assurance

PERMANENT PUBLICATION IDENTITY

Rust Engineering, Memory Safety, and Concurrency Standard

DOCUMENT ID
MYTHOS-SWE-0002

VERSION
1.0.0-candidate

STATUS
Candidate Standard

PUBLISHER
Mythos Systems

PUBLICATION DATE
2026-07-24

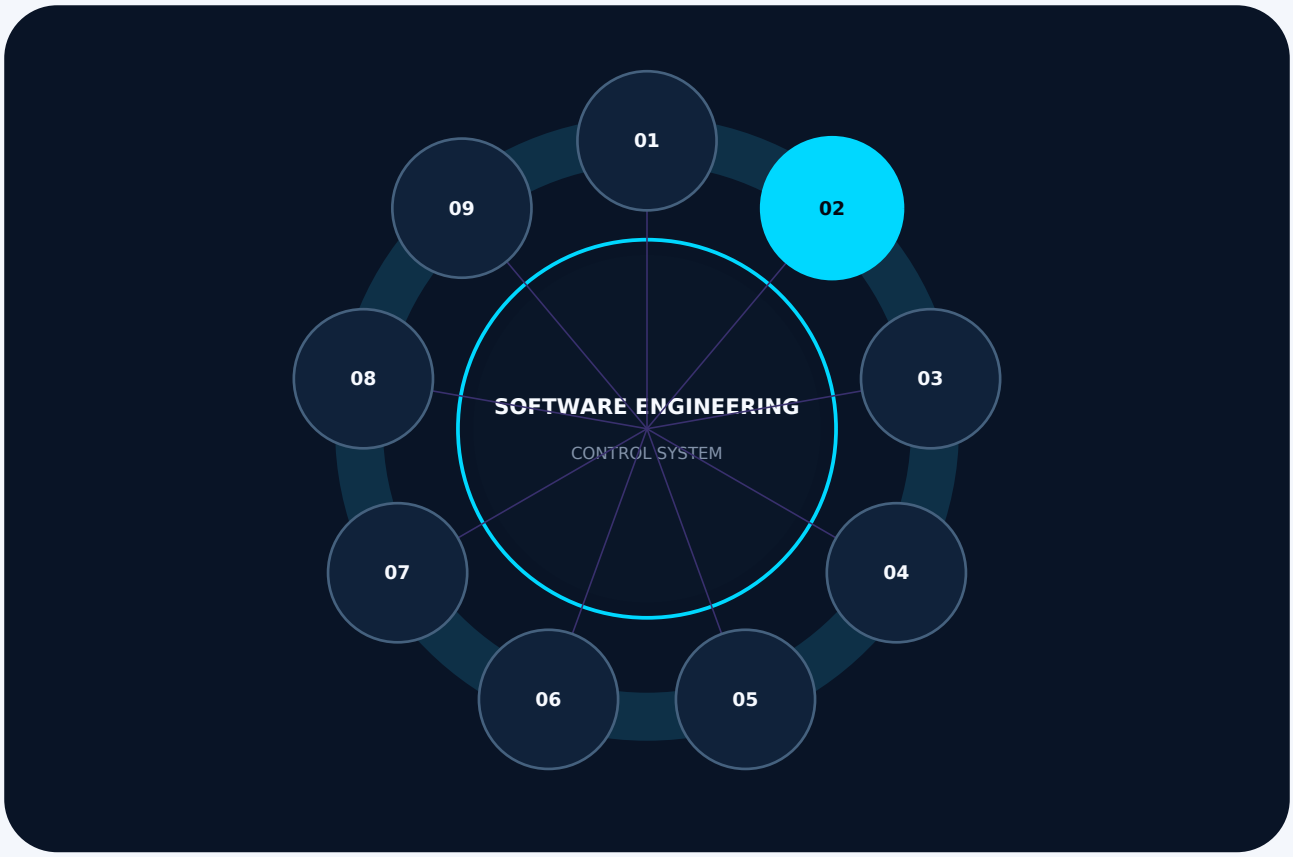
SCHEDULED REVIEW
2027-01-24

15	6	4	1
ATOMIC CRITERIA	ASSURANCE VIEWS	IMPLEMENTATION PROFILES	PERMANENT IDENTITY

Publication doctrine. This publication establishes durable guidance or requirements for its defined scope. Interpretation must preserve the permanent document identity, recorded evidence, controlled exceptions, and formal revision history.

DOMAIN CONTROL SYSTEM

MYTHOS-SWE-0002 inside the software engineering standard constellation



Connected assurance	Architecture, implementation, operations, evidence, and lifecycle are evaluated as one controlled system.
Specialization rule	Companion standards may add precision, but they cannot silently weaken this publication's requirements.

EXECUTIVE POSITION

Rust Engineering, Memory Safety, and Concurrency Standard

The industry's reliance on memory-unsafe languages for critical infrastructure has failed.

Evidence-bearing delivery path

Assurance maturity spectrum

CONTENTS

Publication navigation

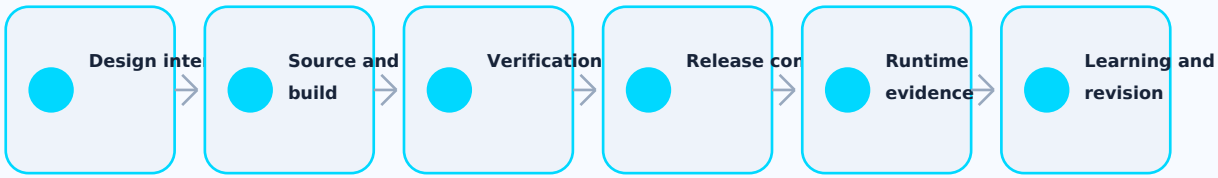
CONTROL TOPOLOGY

Assurance model and evaluation surface

The standard is assessed as an evidence-bearing system. Capability scores do not replace mandatory gates, implementation proof, or operational validation.

Capability family	Evaluation nodes
Memory Safety and `unsafe` Governance (Weight: 25%)	2
Concurrency and Async Discipline (Weight: 20%)	2
Error Handling and Panics (Weight: 20%)	2
Type System and Invariant Enforcement (Weight: 15%)	1
Crate Architecture and Dependencies (Weight: 10%)	1
Tooling and Linting (Weight: 10%)	1
Suite Composition	2
Additional Evaluation Dimension	4

Lifecycle control loop



EVALUATION INDEX

Criteria and evidence map

#	Capability family	Criterion
1	Memory Safety and `unsafe` Governance (Weight: 25%)	`unsafe` Quarantine
2	Memory Safety and `unsafe` Governance (Weight: 25%)	Memory Allocation Discipline
3	Concurrency and Async Discipline (Weight: 20%)	Bounded Concurrency
4	Concurrency and Async Discipline (Weight: 20%)	Cancellation and Shutdown
5	Error Handling and Panics (Weight: 20%)	Panic-Free Execution Paths
6	Error Handling and Panics (Weight: 20%)	Typed Error Architecture
7	Type System and Invariant Enforcement (Weight: 15%)	Type-State Modeling
8	Crate Architecture and Dependencies (Weight: 10%)	Dependency Minimalism
9	Tooling and Linting (Weight: 10%)	Clippy and Formatting
10	Suite Composition	MRES-Safety
11	Suite Composition	MRES-Concurrency
12	Additional Evaluation Dimension	Rust Dimensions
13	Additional Evaluation Dimension	The Rust Doctrine
14	Additional Evaluation Dimension	Scoring Model
15	Additional Evaluation Dimension	Suite Composition

NORMATIVE PUBLICATION

Requirements, evaluation methodology, and implementation guidance

The following body preserves the substantive source standard while normalizing publication status, identity, navigation, and visual presentation.

Executive Mandate

The industry's reliance on memory-unsafe languages for critical infrastructure has failed.

Over 70% of all severe CVEs (buffer overflows, use-after-free, double-free) stem from memory corruption in C and C++ codebases. Simultaneously, garbage-collected languages (Go, Java) trade deterministic performance and memory safety for unpredictable latency (GC pauses) and hidden concurrency bugs (data races masked by mutexes). For AI agent execution planes and deterministic infrastructure runtimes, a GC pause is an outage, and a segfault is a security breach.

This standard exists because:

1. A Segfault is a Security Vulnerability: Memory corruption is not just a crash; it is the primary attack vector for privilege escalation. Rust's ownership model eliminates this class of bugs at compile time.
2. A Data Race is a Silent Corruption: Concurrent mutation without strict ownership rules leads to heisenbugs that are impossible to reproduce. Rust's `Send` and `Sync` traits guarantee thread safety at compile time.
3. `unsafe` is a Privilege, Not a Right: While Rust allows `unsafe` blocks for hardware interaction or FFI, they must be treated like radioactive material: heavily shielded, audited, and quarantined.
4. Panics are Not Error Handling: Using `unwrap()` or `expect()` in a production execution path is an architectural failure. If the runtime can unexpectedly abort, it is not production-ready.

This document establishes the Mythos Rust Engineering Standard (MRES), a comprehensive, evidence-based methodology for evaluating Rust codebases against memory safety, concurrency rigor, and operational discipline.

Scope and Applicability

In Scope

This standard covers the evaluation of:

1. Rust memory safety and ownership models
1. Concurrency patterns (async/await, channels, locks)
1. `unsafe` code governance and auditing
1. Error handling (`Result`, `Option`, typed errors)
1. Crate architecture and dependency management

1. FFI (Foreign Function Interface) boundaries
1. Embedded/WASM targets (`no_std`)

Out of Scope

1. General domain-driven design (covered in MYTHOS-SWE-0001)
1. Polyglot IPC boundaries (covered in MYTHOS-SWE-0005)
1. General supply chain security (covered in MYTHOS-SEC-0002)

Audience

1. Rust engineers and systems programmers
1. Principal architects selecting languages for infrastructure/agent runtimes
1. Security teams evaluating memory safety
1. Platform engineering teams building developer tooling
1. Standards bodies seeking a reference Rust methodology

Definitions and Taxonomy

Rust Dimensions

Dimension	Definition
Ownership	Rust's core memory management model where each value has a single owner, and when the owner goes out of scope, the value is dropped.
`Send`/`Sync`	Marker traits that guarantee a type can be safely transferred across thread boundaries (`Send`) or shared by references (`Sync`).
`unsafe`	A keyword that opts out of Rust's safety guarantees, allowing raw pointer dereferencing, FFI calls, and mutable statics.
Bounded Concurrency	Limiting the number of concurrent tasks or requests to prevent resource exhaustion (using bounded channels, semaphores).
Type-State	A pattern where the state of an object is encoded in its type, making invalid states unrepresentable at compile time.

The Rust Doctrine

> A segfault is a security vulnerability. A data race is a silent corruption. `unwrap()` is a loaded gun. `unsafe` is a privilege, not a right. If it compiles, it must not corrupt memory.

Evaluation Methodology

Scoring Model

Each capability is scored from 0 to 5.

Score	Meaning
0	Fails basic task; rampant `unsafe`, unbounded concurrency, panics in prod
1	Basic Rust but lacks strict error handling or `unsafe` governance
2	Functional but lacks bounded concurrency or type-state modeling
3	Solid production performance; zero `unsafe` in domain, bounded async
4	Strong performance; formal `unsafe` audits, type-state, backpressure
5	Best-in-class; sets the standard for formally verified, memory-safe systems

Weighting

Category	Weight	Rationale
Memory Safety & `unsafe` Governance	25%	Memory corruption is the #1 security threat
Concurrency & Async Discipline	20%	Unbounded concurrency causes outages
Error Handling & Panics	20%	Panics in production paths are unacceptable
Type System & Invariant Enforcement	15%	The type system must make invalid states unrepresentable
Crate Architecture & Dependencies	10%	Dependency sprawl introduces supply chain risk
Tooling & Linting	10%	Strict Clippy and formatting are mandatory

Total: 100%

A codebase MUST score at least 3.0 in Memory Safety and Error Handling to be considered for production use.

Evaluation Categories

Memory Safety and `unsafe` Governance (Weight: 25%)

`unsafe` Quarantine

Is `unsafe` code isolated, audited, and minimized?

Score	Criteria
0	`unsafe` scattered throughout business logic
1	`unsafe` exists but is not documented
2	`unsafe` isolated to specific modules but no audit process
3	`unsafe` encapsulated in safe abstractions with safety comments (`// SAFETY: ...`)
4	`unsafe` requires PR approval from a designated "unsafe auditor"; `cargo geiger` monitored
5	Perfect governance: `unsafe` strictly limited to FFI/hardware boundaries, formally verified safe wrappers, Miri-tested, zero unreviewed `unsafe`

Memory Allocation Discipline

Are allocations predictable and bounded?

Score	Criteria
0	Hidden allocations in hot loops; frequent `clone()` to satisfy the borrow checker
1	Awareness of allocation but no strict bounds
2	Use of arenas or pool allocators for known patterns
3	Zero-allocation parsing/processing in critical paths
4	Custom allocators with OOM (Out-of-Memory) handling that doesn't abort
5	Perfect discipline: formally verified memory bounds, `no_std` capable, zero unexpected allocations

Concurrency and Async Discipline (Weight: 20%)

Bounded Concurrency

Are all async tasks and channels bounded to prevent resource exhaustion?

Score	Criteria
0	Unbounded channels (`mpsc::unbounded`); spawning tasks without limits
1	Bounded channels exist but semaphores not used for task spawning
2	`tokio::sync::Semaphore` or similar used for task bounding
3	All channels bounded; backpressure explicitly handled at system edges
4	Bounded concurrency + graceful degradation when limits hit
5	Perfect discipline: mathematically proven concurrency limits, formally verified deadlock freedom, zero unbounded queues

Cancellation and Shutdown

Does the system support graceful, deterministic shutdown?

Score	Criteria
0	Tasks cannot be cancelled; `tokio::spawn` without `JoinHandle`
1	`tokio::select!` used for cancellation but no cleanup
2	`CancellationToken` used; resources cleaned up on cancel
3	Structured concurrency; tasks don't outlive their parents
4	Deterministic shutdown sequences; timeout-enforced graceful periods
5	Perfect shutdown: formally verified state cleanup, zero task leaks, zero data loss on cancellation

Error Handling and Panics (Weight: 20%)

Panic-Free Execution Paths

Are panics (`unwrap()`, `expect()`, index out of bounds) eliminated from production code paths?

Score	Criteria
0	Frequent `unwrap()` in production paths
1	`expect()` used with messages but still panicking

Score	Criteria
2	`unwrap()` only in tests; `?` operator used in production
3	Zero `unwrap()`/`expect()` in production paths; all errors typed
4	`#![deny(clippy::unwrap_used)]` enforced in CI
5	Perfect handling: zero panics possible, formally verified error propagation, catch_unwind only at FFI boundaries

Typed Error Architecture

Are errors structured and distinguishable?

Score	Criteria
0	`Box` everywhere
1	`anyhow` for applications, but no domain error types
2	`thiserror` for library crates; domain errors defined
3	Errors classified as Retryable vs. Fatal
4	Error chains preserve context; structured logging of errors
5	Perfect architecture: typed error hierarchy, retryable classification, structured envelopes, no sensitive data in errors

Type System and Invariant Enforcement (Weight: 15%)

Type-State Modeling

Are invalid states made unrepresentable at compile time?

Score	Criteria
0	Runtime checks for state validity (e.g., `if state == "connected"` panic)
1	Enums used for state but transitions not enforced
2	Type-state pattern used for critical state machines (e.g., `Uninitialized -> Connected`)
3	Builder pattern with consume-by-move ensuring mandatory fields
4	Comprehensive type-state for all protocol/lifecycle states
5	Perfect enforcement: zero runtime state validation needed, compile-time guaranteed valid transitions

Crate Architecture and Dependencies (Weight: 10%)

Dependency Minimalism

Is the dependency tree lean and audited?

Score	Criteria
0	Heavy dependency tree; unnecessary crates pulled in
1	`cargo-deny` run manually
2	`cargo-deny` in CI for license and security advisories
3	Strict dependency review; no optional features pulled in blindly

Score	Criteria
4	Workspace architecture isolates dependencies to specific crates
5	Perfect minimalism: `cargo-vet` or formal audit, zero transitive vulnerabilities, minimal crate footprint

Tooling and Linting (Weight: 10%)

Clippy and Formatting

Are strict lints enforced in CI?

Score	Criteria
0	No Clippy or rustfmt enforcement
1	Default Clippy warnings
2	`clippy::all` as warning
3	`clippy::all` + `clippy::pedantic` as deny
4	Custom clippy configuration enforcing project-specific rules (e.g., deny `unwrap_used`)
5	Perfect enforcement: zero warnings, pedantic lints, automated formatting, custom lint rules for domain invariants

The Mythos Rust Engineering Suite (MRES)

Traditional benchmarks measure build time. The MRES evaluates memory safety, concurrency bounds, and panic elimination.

Suite Composition

MRES-Safety

1. Panic Detection: 100+ tests scanning codebase for `unwrap()`/`expect()` in non-test paths.
1. Unsafe Audit: 50+ tests verifying `unsafe` blocks have `// SAFETY:` comments and are encapsulated.

MRES-Concurrency

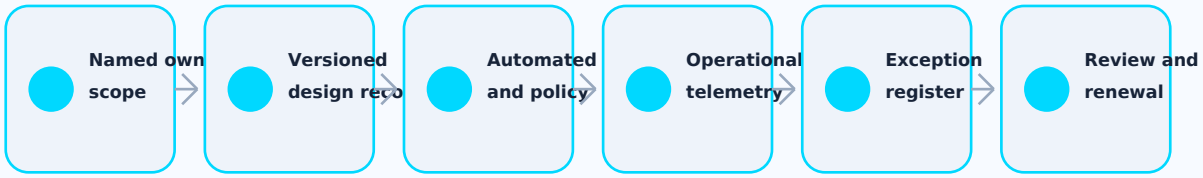
1. Unbounded Queue Detection: 50+ tests scanning for unbounded channels.
1. Shutdown Race: 50+ tests abruptly cancelling tasks to verify no resource leaks or deadlocks.

Execution Protocol

ASSURANCE AND ADOPTION

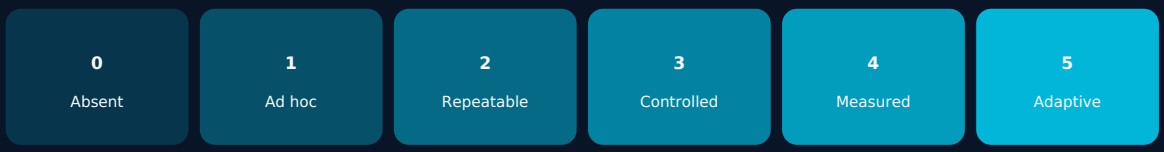
Implementation evidence and review gates

Required assurance chain



A passing score is not sufficient without current evidence. High-assurance adoption requires traceable design decisions, reproducible builds, independently reviewable tests, controlled secrets, runtime observability, failure exercises, and a dated exception register.

Assurance maturity spectrum



Current authority register

Authority	Version / snapshot	Applicability	Review by
Rust Project: The Rust Programming Language and Rust Reference	Rolling official documentation; book assumes Rust 1.90.0 or later and Rust 2024 Edition	High	2026-10-22
Go Project: The Go Programming Language Specification and Effective Go	Rolling official documentation snapshot	High	2026-10-22
Python Software Foundation: Python Language Reference, Packaging, Typing, and PEP Index	Rolling official documentation snapshot	High	2026-10-22
Tauri Programme: Tauri 2 Documentation	Tauri 2 rolling documentation snapshot	High	2026-10-22
Svelte Project: Svelte Documentation	Svelte 5 rolling documentation snapshot	High	2026-10-22
TLA+ Foundation: TLA+ Language and Tools Documentation	Rolling documentation snapshot	High	2027-01-24
OpenAPI Initiative: OpenAPI Specification	3.2.0	High	2027-01-24
OpenTelemetry Project: OpenTelemetry Documentation and Specification	Rolling specification snapshot; profiles public alpha and declarative configuration stable in 2026	High	2026-10-22

Research limitation. Official documentation establishes published interfaces and declared behavior. It does not prove the performance, security, interoperability, cost, or operational suitability of a specific implementation. Those claims require controlled testing and retained evidence.

ENGINEERING STANDARDS LIBRARY / SOFTWARE ENGINEERING

Go Engineering, Concurrency, and Service Architecture Standard

Simple service boundaries, goroutine lifecycle discipline, backpressure, cancellation, observability, and operable Go systems.

PERMANENT DOCUMENT ID

MYTHOS-SWE-0003

PUBLICATION

Candidate Standard
Version 1.0.0-candidate

ASSURANCE PROFILE

Standard / Advanced
High Assurance

PERMANENT PUBLICATION IDENTITY

Go Engineering, Concurrency, and Service Architecture Standard

DOCUMENT ID
MYTHOS-SWE-0003

VERSION
1.0.0-candidate

STATUS
Candidate Standard

PUBLISHER
Mythos Systems

PUBLICATION DATE
2026-07-24

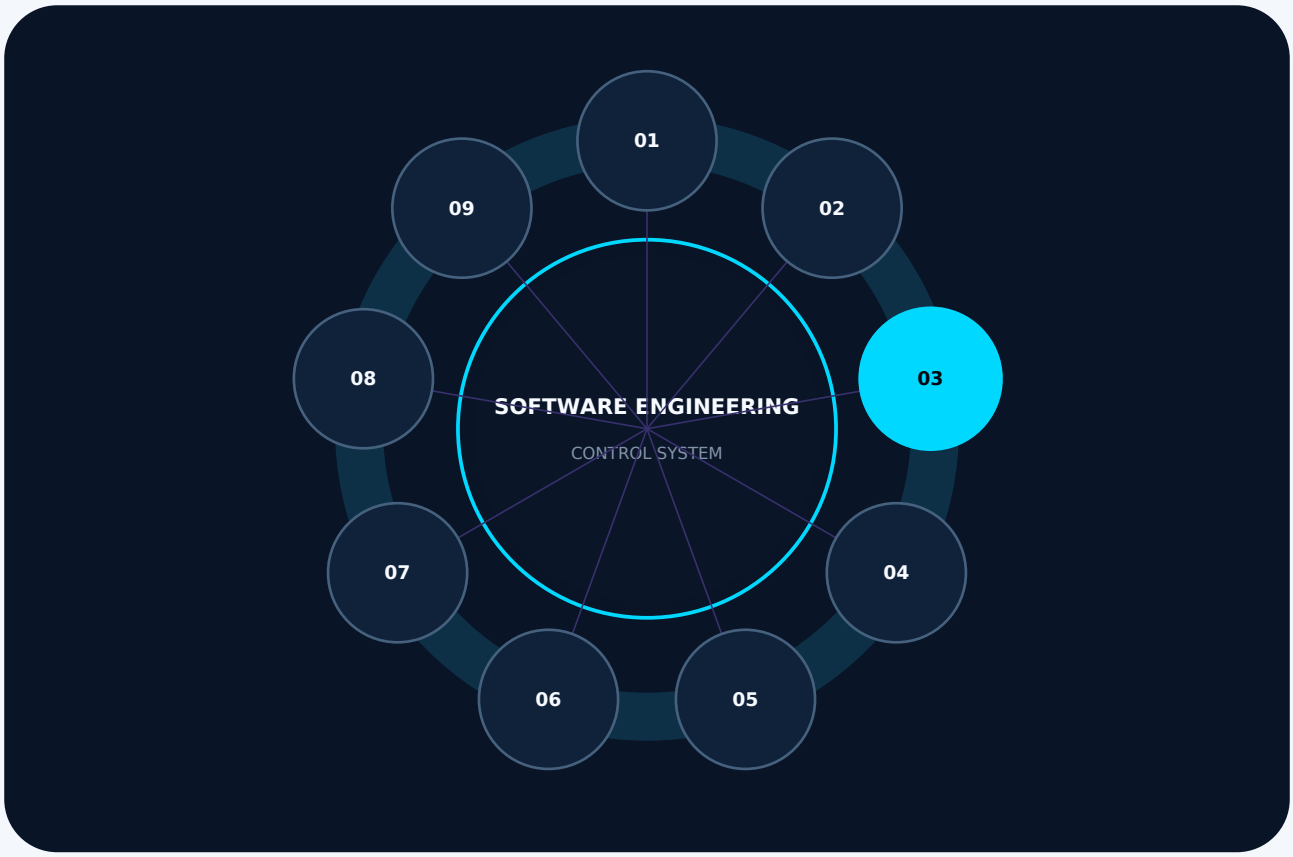
SCHEDULED REVIEW
2027-01-24

15	6	4	1
ATOMIC CRITERIA	ASSURANCE VIEWS	IMPLEMENTATION PROFILES	PERMANENT IDENTITY

Publication doctrine. This publication establishes durable guidance or requirements for its defined scope. Interpretation must preserve the permanent document identity, recorded evidence, controlled exceptions, and formal revision history.

DOMAIN CONTROL SYSTEM

MYTHOS-SWE-0003 inside the software engineering standard constellation



Connected assurance	Architecture, implementation, operations, evidence, and lifecycle are evaluated as one controlled system.
Specialization rule	Companion standards may add precision, but they cannot silently weaken this publication's requirements.

EXECUTIVE POSITION

Go Engineering, Concurrency, and Service Architecture Standard

The deployment of Go services in production has failed to live up to the language's promise of simple, reliable concurrency.

Evidence-bearing delivery path

Assurance maturity spectrum

CONTENTS

Publication navigation

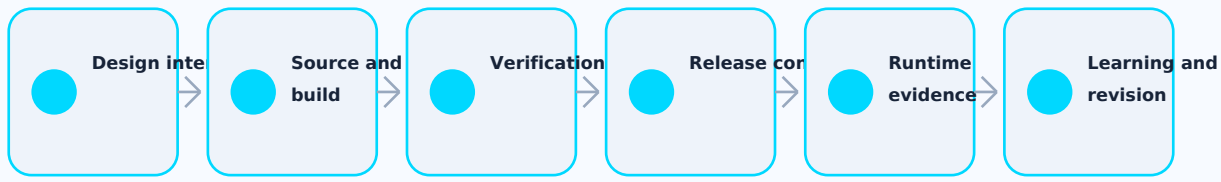
CONTROL TOPOLOGY

Assurance model and evaluation surface

The standard is assessed as an evidence-bearing system. Capability scores do not replace mandatory gates, implementation proof, or operational validation.

Capability family	Evaluation nodes
Goroutine Lifecycle and Concurrency (Weight: 25%)	2
Error Handling and Observability (Weight: 25%)	2
Context Discipline (Weight: 15%)	1
Interface and Package Architecture (Weight: 15%)	2
Race Conditions and Safety (Weight: 10%)	1
Operational Lifecycle (Weight: 10%)	1
Suite Composition	2
Additional Evaluation Dimension	4

Lifecycle control loop



EVALUATION INDEX

Criteria and evidence map

#	Capability family	Criterion
1	Goroutine Lifecycle and Concurrency (Weight: 25%)	Bounded Concurrency
2	Goroutine Lifecycle and Concurrency (Weight: 25%)	Goroutine Lifecycle Tracking
3	Error Handling and Observability (Weight: 25%)	Error Wrapping
4	Error Handling and Observability (Weight: 25%)	Structured Logging
5	Context Discipline (Weight: 15%)	Context Propagation
6	Interface and Package Architecture (Weight: 15%)	Interface Segregation
7	Interface and Package Architecture (Weight: 15%)	Package Structure
8	Race Conditions and Safety (Weight: 10%)	Race Detector
9	Operational Lifecycle (Weight: 10%)	Graceful Shutdown
10	Suite Composition	M-GO-S-Lifecycle
11	Suite Composition	M-GO-S-Errors
12	Additional Evaluation Dimension	Go Dimensions
13	Additional Evaluation Dimension	The Go Doctrine
14	Additional Evaluation Dimension	Scoring Model
15	Additional Evaluation Dimension	Suite Composition

NORMATIVE PUBLICATION

Requirements, evaluation methodology, and implementation guidance

The following body preserves the substantive source standard while normalizing publication status, identity, navigation, and visual presentation.

Executive Mandate

The deployment of Go services in production has failed to live up to the language's promise of simple, reliable concurrency.

While Go's goroutines and channels make concurrent code easy to write, they make it exceptionally difficult to reason about lifecycle, resource exhaustion, and error propagation. Organizations deploy Go services that silently leak goroutines, swallow context cancellations, and return untraceable `error` interfaces up the stack, turning distributed tracing into a black box.

This standard exists because:

1. Goroutine Leaks are Silent Killers: A goroutine that never terminates holds onto memory, stack space, and referenced closures. In a long-running service, a leak of 1 goroutine per request will exhaust memory and crash the process. Goroutine lifecycle MUST be as rigorously managed as memory in C.
2. `if err != nil { return err }` is Anti-Observability: Returning bare errors destroys the stack trace and context. Production debugging requires structured, wrapped errors with domain context, correlation IDs, and classification (retryable vs. fatal).
3. Context Cancellation is Not Optional: Ignoring `ctx.Done()` in I/O-bound goroutines creates zombie processes that consume resources even after the client has disconnected. All blocking operations MUST respect context.
4. Unbounded Concurrency is a DoS Vector: Spawning a goroutine per request without a semaphore or bounded channel is equivalent to disabling backpressure. The system WILL crash under load.

This document establishes the Mythos Go Engineering Standard (MGES), a comprehensive, evidence-based methodology for evaluating Go codebases against concurrency safety, error discipline, and service architecture.

Scope and Applicability

In Scope

This standard covers the evaluation of:

1. Goroutine lifecycle management and leak prevention
1. Structured error handling and wrapping (`fmt.Errorf`, custom types)
1. `context.Context` propagation and cancellation

1. Bounded concurrency (semaphores, worker pools, bounded channels)
1. Interface design and package architecture
1. Race condition prevention and `-race` enforcement
1. Memory management and GC tuning for low-latency services
1. Service initialization and graceful shutdown

Out of Scope

1. General domain-driven design (covered in MYTHOS-SWE-0001)
1. Rust memory safety (covered in MYTHOS-SWE-0002)
1. Polyglot IPC boundaries (covered in MYTHOS-SWE-0005)

Audience

1. Go engineers and systems programmers
1. Principal architects selecting languages for control planes
1. Security teams evaluating concurrency safety
1. Platform engineering teams building Go service scaffolding
1. Standards bodies seeking a reference Go methodology

Definitions and Taxonomy

Go Dimensions

Dimension	Definition
Goroutine Leak	A goroutine that blocks forever or runs indefinitely without a termination condition, preventing the garbage collector from reclaiming its stack and referenced memory.
Context Propagation	The practice of passing `context.Context` as the first parameter to all functions performing I/O or long-running work, enabling cancellation and deadlines.
Bounded Concurrency	Limiting the number of concurrent goroutines using semaphores (`chan struct{}`), worker pools, or `errgroup.Group` with `SetLimit()`.
Error Wrapping	The practice of adding context to an error using `fmt.Errorf("doing X: %w", err)` or custom error types, preserving the original error chain for `errors.Is` and `errors.As`.

The Go Doctrine

> A goroutine without a lifecycle is a leak. An error without context is a lie. A context without cancellation is a zombie. Simplicity is not an excuse for negligence.

Evaluation Methodology

Scoring Model

Each capability is scored from 0 to 5.

Score	Meaning
0	Fails basic task; goroutine leaks, bare errors, unbounded concurrency
1	Basic context usage but lacks error wrapping or lifecycle management
2	Functional but lacks bounded concurrency or structured errors
3	Solid production performance; bounded, wrapped, context-aware
4	Strong performance; structured error classification, leak detection
5	Best-in-class; sets the standard for deterministic, observable Go services

Weighting

Category	Weight	Rationale
Goroutine Lifecycle & Concurrency	25%	Goroutine leaks are the #1 operational failure in Go
Error Handling & Observability	25%	Bare errors make debugging impossible
Context Discipline	15%	Ignored contexts cause resource exhaustion
Interface & Package Architecture	15%	Circular dependencies and "util" packages kill modularity
Race Conditions & Safety	10%	Map concurrent writes cause panics
Operational Lifecycle	10%	Graceful shutdown is mandatory

Total: 100%

A codebase MUST score at least 3.0 in Goroutine Lifecycle and Error Handling to be considered for production use.

Evaluation Categories

Goroutine Lifecycle and Concurrency (Weight: 25%)

Bounded Concurrency

Are all goroutine spawns bounded and backpressured?

Score	Criteria
0	`go func()` spawned per request without limits
1	Bounded channels used but semaphores not
2	`errgroup.Group` used for fan-out, but unbounded
3	`errgroup.Group` with `SetLimit()` or semaphore for all fan-out
4	Bounded concurrency + graceful degradation when limits hit

Score	Criteria
5	Perfect discipline: formally verified concurrency limits, automated leak detection in CI (`goleak`), zero unbounded spawns

Goroutine Lifecycle Tracking

Can the system prove no goroutines are leaked?

Score	Criteria
0	No tracking; rely on monitoring memory growth
1	Manual code review for leaks
2	`go.uber.org/goleak` used in tests
3	All goroutines have explicit `select { case <-ctx.Done(): ... }` exit conditions
4	Runtime metrics (`runtime.NumGoroutine`) monitored and alerted
5	Perfect tracking: CI-enforced `goleak`, formally verified exit conditions, zero `go func()` without context

Error Handling and Observability (Weight: 25%)

Error Wrapping

Are errors wrapped with domain context before returning?

Score	Criteria
0	Bare `return err` everywhere
1	`fmt.Errorf("message: %v", err)` (loses wrapping)
2	`fmt.Errorf("message: %w", err)` (preserves chain)
3	Custom error types with domain context and `Unwrap()`
4	Error classification (Retryable, Permanent, NotFound) using `errors.Is`/`errors.As`
5	Perfect handling: structured error envelopes, classification, redaction, and zero bare `return err` outside standard library

Structured Logging

Are errors logged with structured context (trace ID, tenant, request ID)?

Score	Criteria
0	`fmt.Println` or `log.Printf`
1	`logrus` or `zap` but unstructured messages
2	Structured logging (`slog`/`zap`) with key-value pairs
3	Correlation IDs and trace IDs injected into context and logged
4	Errors logged with structured stack traces and domain state
5	Perfect observability: structured logging, redaction of secrets, `slog` context propagation, and zero unstructured error logs

Context Discipline (Weight: 15%)

Context Propagation

Do all I/O and blocking operations accept and respect `context.Context`?

Score	Criteria
0	Functions perform I/O without context
1	Context created at function start, not propagated from caller
2	Context passed but ignored in blocking operations
3	All I/O operations accept `ctx` and pass it down
4	All blocking `select` statements include `case <-ctx.Done()`
5	Perfect discipline: no I/O without context, all goroutines respect cancellation, enforced by linter

Interface and Package Architecture (Weight: 15%)

Interface Segregation

Are interfaces small, defined by the consumer, and not "dumb" data bags?

Score	Criteria
0	God interfaces with 20+ methods
1	Interfaces defined at the provider, not consumer
2	Consumer-defined interfaces (Go style) but still large
3	Single-method interfaces (`io.Reader`, `io.Writer` style) prevalent
4	Interfaces used to decouple from external dependencies (databases, APIs)
5	Perfect architecture: consumer-defined, single-responsibility interfaces, zero provider-defined interfaces outside `internal`

Package Structure

Is the package structure modular and free of circular dependencies?

Score	Criteria
0	Everything in `package main` or a single `utils` package
1	Packages organized by layer (controllers, services, models)
2	Packages organized by domain/bounded context
3	Use of `internal` to prevent external dependencies on implementation
4	Clean dependency graph; no circular dependencies; `cmd/` separate from `internal/`
5	Perfect structure: formally verified dependency graph, domain-centric, zero `util` or `helpers` packages

Race Conditions and Safety (Weight: 10%)

Race Detector

Is the Go race detector (`-race`) mandatory in CI and tests?

Score	Criteria
0	`-race` never used
1	`-race` used locally sometimes
2	`-race` used in unit tests only
3	`-race` mandatory in CI for all tests
4	`-race` used in integration tests and load tests
5	Perfect enforcement: `-race` mandatory in all CI stages, zero map concurrent writes, mutexes validated

Operational Lifecycle (Weight: 10%)

Graceful Shutdown

Does the service handle `SIGTERM` and drain connections?

Score	Criteria
0	Immediate exit on `SIGTERM`; requests dropped
1	`os.Signal` caught but no drain logic
2	HTTP server `Shutdown()` with a timeout
3	Graceful shutdown of all goroutines using `context.WithTimeout`
4	Shutdown sequence respects in-flight requests and completes DB transactions
5	Perfect shutdown: deterministic drain, formally verified in-flight completion, zero dropped requests

The Mythos Go Engineering Suite (M-GO-S)

Traditional benchmarks measure p99 latency. The M-GO-S evaluates goroutine leaks, error context, and concurrency bounds.

Suite Composition

M-GO-S-Lifecycle

1. Goroutine Leak: 100+ tests running `go.uber.org/goleak` after every test suite.
1. Context Cancel: 50+ tests cancelling context mid-request, verifying no goroutines hang.

M-GO-S-Errors

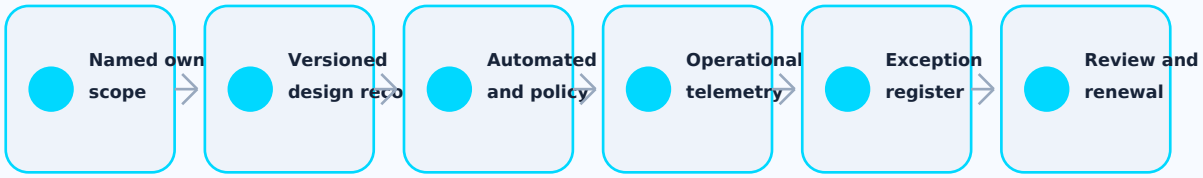
1. Bare Error Scan: 100+ tests scanning for `return err` without wrapping.
1. Structured Log Check: 50+ tests verifying errors are logged with trace IDs.

Execution Protocol

ASSURANCE AND ADOPTION

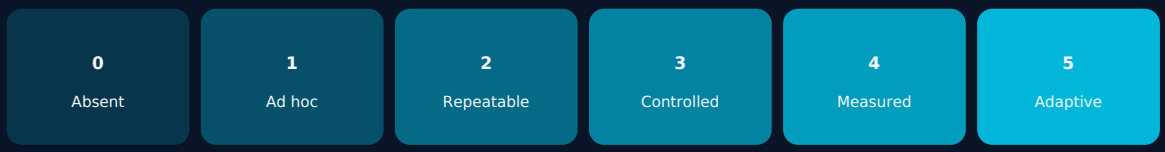
Implementation evidence and review gates

Required assurance chain



A passing score is not sufficient without current evidence. High-assurance adoption requires traceable design decisions, reproducible builds, independently reviewable tests, controlled secrets, runtime observability, failure exercises, and a dated exception register.

Assurance maturity spectrum



Current authority register

Authority	Version / snapshot	Applicability	Review by
Rust Project: The Rust Programming Language and Rust Reference	Rolling official documentation; book assumes Rust 1.90.0 or later and Rust 2024 Edition	High	2026-10-22
Go Project: The Go Programming Language Specification and Effective Go	Rolling official documentation snapshot	High	2026-10-22
Python Software Foundation: Python Language Reference, Packaging, Typing, and PEP Index	Rolling official documentation snapshot	High	2026-10-22
Tauri Programme: Tauri 2 Documentation	Tauri 2 rolling documentation snapshot	High	2026-10-22
Svelte Project: Svelte Documentation	Svelte 5 rolling documentation snapshot	High	2026-10-22
TLA+ Foundation: TLA+ Language and Tools Documentation	Rolling documentation snapshot	High	2027-01-24
OpenAPI Initiative: OpenAPI Specification	3.2.0	High	2027-01-24
OpenTelemetry Project: OpenTelemetry Documentation and Specification	Rolling specification snapshot; profiles public alpha and declarative configuration stable in 2026	High	2026-10-22

Research limitation. Official documentation establishes published interfaces and declared behavior. It does not prove the performance, security, interoperability, cost, or operational suitability of a specific implementation. Those claims require controlled testing and retained evidence.

ENGINEERING STANDARDS LIBRARY / SOFTWARE ENGINEERING

Python Engineering, Type Safety, and Automation Standard

Typed Python, deterministic automation, dependency control, testability, packaging, and maintainable scripting at enterprise scale.

PERMANENT DOCUMENT ID

MYTHOS-SWE-0004

PUBLICATION

Candidate Standard
Version 1.0.0-candidate

ASSURANCE PROFILE

Standard / Advanced
High Assurance

PERMANENT PUBLICATION IDENTITY

Python Engineering, Type Safety, and Automation Standard

DOCUMENT ID
MYTHOS-SWE-0004

VERSION
1.0.0-candidate

STATUS
Candidate Standard

PUBLISHER
Mythos Systems

PUBLICATION DATE
2026-07-24

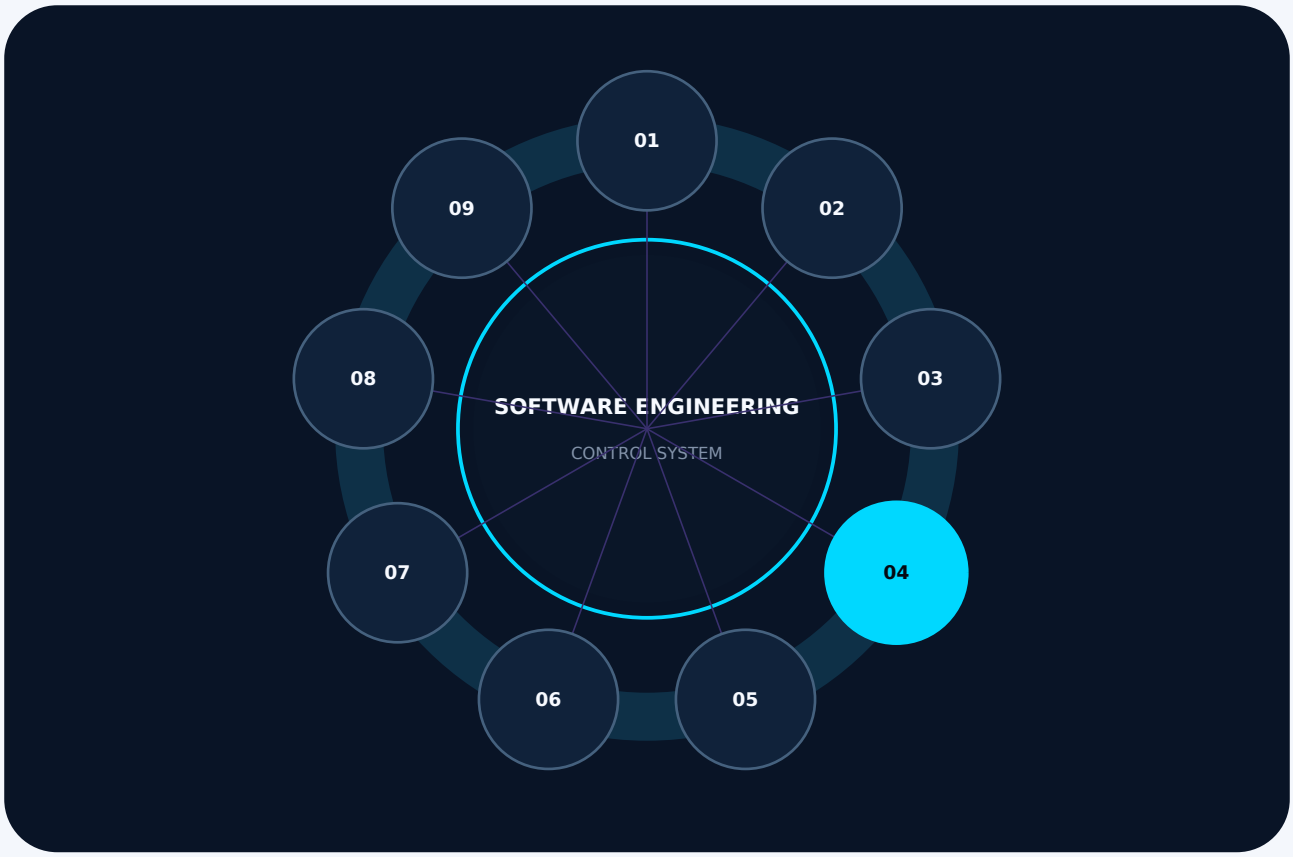
SCHEDULED REVIEW
2027-01-24

16	6	4	1
ATOMIC CRITERIA	ASSURANCE VIEWS	IMPLEMENTATION PROFILES	PERMANENT IDENTITY

Publication doctrine. This publication establishes durable guidance or requirements for its defined scope. Interpretation must preserve the permanent document identity, recorded evidence, controlled exceptions, and formal revision history.

DOMAIN CONTROL SYSTEM

MYTHOS-SWE-0004 inside the software engineering standard constellation



Connected assurance	Architecture, implementation, operations, evidence, and lifecycle are evaluated as one controlled system.
Specialization rule	Companion standards may add precision, but they cannot silently weaken this publication's requirements.

EXECUTIVE POSITION

Python Engineering, Type Safety, and Automation Standard

The deployment of Python in production infrastructure has failed.

Evidence-bearing delivery path

Assurance maturity spectrum

CONTENTS

Publication navigation

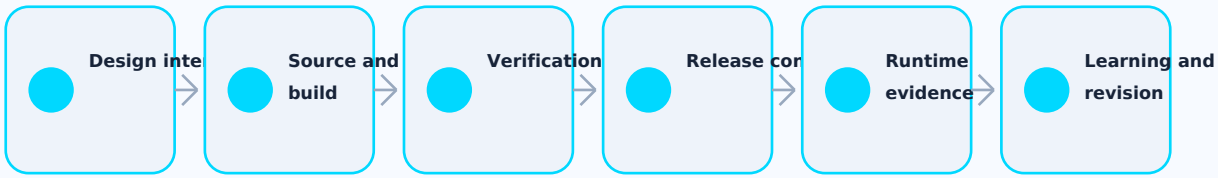
CONTROL TOPOLOGY

Assurance model and evaluation surface

The standard is assessed as an evidence-bearing system. Capability scores do not replace mandatory gates, implementation proof, or operational validation.

Capability family	Evaluation nodes
Type Safety and Strict Mode (Weight: 25%)	2
Data Modeling and Validation (Pydantic) (Weight: 20%)	2
Dependency and Packaging (Weight: 20%)	2
Async and Concurrency (Weight: 15%)	2
Linting and Code Quality (Weight: 10%)	1
Performance and Native Extensions (Weight: 10%)	1
Suite Composition	2
Additional Evaluation Dimension	4

Lifecycle control loop



EVALUATION INDEX

Criteria and evidence map

#	Capability family	Criterion
1	Type Safety and Strict Mode (Weight: 25%)	Static Type Enforcement
2	Type Safety and Strict Mode (Weight: 25%)	`Any` Quarantine
3	Data Modeling and Validation (Pydantic) (Weight: 20%)	Schema Enforcement
4	Data Modeling and Validation (Pydantic) (Weight: 20%)	Serialization Determinism
5	Dependency and Packaging (Weight: 20%)	Dependency Locking
6	Dependency and Packaging (Weight: 20%)	Environment Isolation
7	Async and Concurrency (Weight: 15%)	Structured Concurrency
8	Async and Concurrency (Weight: 15%)	CPU-Bound Offloading
9	Linting and Code Quality (Weight: 10%)	Linting and Formatting
10	Performance and Native Extensions (Weight: 10%)	Performance Profiling
11	Suite Composition	M-PY-S-Types
12	Suite Composition	M-PY-S-Models
13	Additional Evaluation Dimension	Python Dimensions
14	Additional Evaluation Dimension	The Python Doctrine
15	Additional Evaluation Dimension	Scoring Model
16	Additional Evaluation Dimension	Suite Composition

NORMATIVE PUBLICATION

Requirements, evaluation methodology, and implementation guidance

The following body preserves the substantive source standard while normalizing publication status, identity, navigation, and visual presentation.

Executive Mandate

The deployment of Python in production infrastructure has failed.

Python is the lingua franca of AI and automation, yet it is treated as a scripting language. Organizations deploy Python agents that accept `Dict[str, Any]` as inputs, rely on runtime duck-typing to catch errors, and manage dependencies via unpinned `requirements.txt` files. This results in runtime `AttributeError` and `KeyError` exceptions that crash automation pipelines, hallucinate infrastructure configurations, and exfiltrate data through unvalidated schemas.

This standard exists because:

1. A Runtime TypeError is an Architectural Failure: If a type error can reach production, the development pipeline is broken. Python's dynamic nature makes it exceptionally powerful, but also exceptionally dangerous without rigorous static analysis (`mypy`, `pyright`).
2. `Dict[str, Any]` is a Liability: Passing untyped dictionaries through agent pipelines is the root cause of schema drift and silent data corruption. All domain boundaries MUST use strictly typed models (Pydantic).
3. `Any` is a Betrayal: Using `typing.Any` disables the type checker. It is the `unsafe` of Python. It MUST be quarantined and explicitly justified.
4. Automation Code is Production Code: A Python script that pushes a firewall rule has the same blast radius as a compiled C program. It MUST be held to the same rigor: strict typing, locking, testing, and deterministic dependency management.

This document establishes the Mythos Python Engineering Standard (MPES), a comprehensive, evidence-based methodology for evaluating Python codebases against type safety, dependency determinism, and automation rigor.

Scope and Applicability

In Scope

This standard covers the evaluation of:

1. Static type checking (`mypy --strict`, `pyright`)
1. Strict data modeling and validation (Pydantic v2)
1. Dependency management and locking (uv, Poetry, PEP 621)
1. Packaging and distribution (wheel, sdist, PyO3/maturin)

1. Async concurrency and structured concurrency (asyncio, anyio)
1. Code quality and linting (Ruff)
1. Performance profiling and native extensions (Cython, Rust)

Out of Scope

1. General domain-driven design (covered in MYTHOS-SWE-0001)
1. Rust engineering internals (covered in MYTHOS-SWE-0002)
1. General supply chain security (covered in MYTHOS-SEC-0002)

Audience

1. Python engineers and ML/AI developers
1. Principal architects selecting languages for automation/agent planes
1. Security teams evaluating Python supply chain risks
1. Platform engineering teams building Python tooling
1. Standards bodies seeking a reference Python methodology

Definitions and Taxonomy

Python Dimensions

Dimension	Definition
Strict Typing	The enforcement of type hints across the entire codebase using `mypy --strict` or `pyright --strict`, disallowing untyped definitions and `Any`.
Pydantic v2	A data validation library using Rust (pydantic-core) for parsing and validation, ensuring runtime schemas match static types.
Deterministic Dependencies	The practice of locking all dependencies (including transitive) to exact versions and hashes (`uv.lock`, `poetry.lock`) to ensure reproducible builds.
Type-State Modeling	A pattern where the state of an object is encoded in its generic type parameters (e.g., `Agent[State.Uninitialized]`), making invalid states unrepresentable.

The Python Doctrine

> A runtime TypeError is an architectural failure. A dictionary is not a domain model. Any is a betrayal. Automation code is production code. If it runs in production, it must be typed.

Evaluation Methodology

Scoring Model

Each capability is scored from 0 to 5.

Score	Meaning
0	Fails basic task; no typing, unvalidated dicts, unpinned deps

Score	Meaning
1	Basic type hints but not enforced; partial validation
2	Functional but lacks strict mode or Pydantic boundaries
3	Solid production performance; `mypy --strict`, Pydantic v2, locked deps
4	Strong performance; type-state modeling, Rust extensions, Ruff
5	Best-in-class; sets the standard for deterministic, typed Python

Weighting

Category	Weight	Rationale
Type Safety & Strict Mode	25%	Runtime type errors are silent killers
Data Modeling & Validation (Pydantic)	20%	Unvalidated dicts cause schema drift
Dependency & Packaging	20%	Unpinned deps cause supply chain attacks and drift
Async & Concurrency	15%	Blocking I/O kills performance; unbounded async kills availability
Linting & Code Quality	10%	Inconsistent style hides bugs
Performance & Native Extensions	10%	CPU-bound Python blocks the event loop

Total: 100%

A codebase MUST score at least 3.0 in Type Safety and Data Modeling to be considered for production use.

Evaluation Categories

Type Safety and Strict Mode (Weight: 25%)

Static Type Enforcement

Is the codebase strictly typed and enforced in CI?

Score	Criteria
0	No type hints; dynamic typing everywhere
1	Type hints exist but not enforced; `Any` used frequently
2	`mypy` or `pyright` run locally but not in CI
3	`mypy --strict` or `pyright --strict` mandatory in CI
4	Zero `Any` usage without explicit `# type: ignore` with justification
5	Perfect enforcement: strict mode, zero `Any`, type-state modeling, typed generics, no `cast` hacks

`Any` Quarantine

Is `typing.Any` isolated and justified?

Score	Criteria
0	`Any` used as default for complex types
1	`Any` used in external API boundaries only
2	`Any` replaced with `object` or generic `T` where possible
3	`Any` requires code review justification
4	`Any` restricted to deserialization boundaries; immediately validated into typed model
5	Perfect quarantine: zero `Any` in domain logic, formally verified type boundaries

Data Modeling and Validation (Pydantic) (Weight: 20%)

Schema Enforcement

Are all domain boundaries and API contracts validated using strict models?

Score	Criteria
0	`Dict[str, Any]` or raw JSON passed between functions
1	`dataclasses` used but no runtime validation
2	Pydantic models used for API boundaries
3	Pydantic v2 with `strict=True` mode globally
4	Custom Pydantic types for domain primitives (e.g., `IPv4Address`, `SVID`)
5	Perfect enforcement: Pydantic v2 strict, type-state modeling via generics, zero unvalidated dicts in codebase

Serialization Determinism

Are serialization/deserialization deterministic and versioned?

Score	Criteria
0	`json.loads` into dict; no schema
1	Pydantic serialization but no versioning
2	Versioned schemas via model validators
3	Backward/forward compatibility tested
4	Deterministic serialization (sorted keys, explicit excludes)
5	Perfect determinism: formally verified schema evolution, zero breaking changes, cryptographic hashing of serialized state

Dependency and Packaging (Weight: 20%)

Dependency Locking

Are dependencies strictly locked and hashed?

Score	Criteria
0	`requirements.txt` without versions
1	Pinned versions in `requirements.txt`
2	`poetry.lock` or `uv.lock` committed

Score	Criteria
3	Lock file includes transitive dependencies and hashes
4	Automated dependency scanning (SBOM generation) in CI
5	Perfect locking: `uv` with strict hashes, air-gapped registry support, zero floating versions, automated CVE blocking

Environment Isolation

Are Python environments deterministic and isolated?

Score	Criteria
0	Global Python install; `pip install` locally
1	`virtualenv` used manually
2	`direnv` or `venv` automated
3	`uv` for fast, deterministic environment creation
4	Docker images with multi-stage builds; no build tools in runtime
5	Perfect isolation: `uv`, reproducible builds, Nix-like determinism, zero "works on my machine"

Async and Concurrency (Weight: 15%)

Structured Concurrency

Is async code structured and bounded?

Score	Criteria
0	Synchronous blocking I/O in services
1	`asyncio` used but unstructured `asyncio.create_task`
2	`anyio` or `asyncio.TaskGroup` (Python 3.11+) used for structured concurrency
3	Bounded semaphores for concurrent operations
4	Cancellation explicitly handled; resources cleaned up on `CancelledError`
5	Perfect discipline: structured concurrency, bounded, cancellation-safe, zero leaked tasks

CPU-Bound Offloading

Are CPU-bound tasks offloaded to avoid blocking the event loop?

Score	Criteria
0	CPU-intensive parsing/computation in async functions
1	`asyncio.to_thread` used for blocking calls
2	`ProcessPoolExecutor` for CPU-bound work
3	Rust extensions (PyO3) for performance-critical paths
4	Subprocess isolation for untrusted code execution
5	Perfect offloading: Rust/compiled extensions for hot paths, zero event loop blocking

Linting and Code Quality (Weight: 10%)

Linting and Formatting

Is the codebase consistently formatted and linted?

Score	Criteria
0	No linters or formatters
1	`flake8` and `black` used inconsistently
2	`ruff` used for linting and formatting in CI
3	`ruff` with strict rule set (e.g., `ALL` with specific ignores)
4	Custom `ruff` rules enforcing project standards
5	Perfect quality: zero lint warnings, automated formatting, custom rules enforcing type safety and domain invariants

Performance and Native Extensions (Weight: 10%)

Performance Profiling

Are performance bottlenecks identified and resolved with compiled code?

Score	Criteria
0	No profiling; blind optimization
1	`cProfile` used occasionally
2	Continuous profiling in staging (e.g., `py-spy`, `austin`)
3	Hot paths identified and optimized with pure Python
4	Cython or C-extensions for critical paths
5	Perfect performance: PyO3/Rust extensions for all hot paths, formally benchmarked, zero GIL contention

The Mythos Python Suite (M-PY-S)

Traditional Python benchmarks measure test coverage. The M-PY-S evaluates type strictness, schema validation, and dependency determinism.

Suite Composition

M-PY-S-Types

1. Strict Mode: 100+ tests running `mypy --strict` with zero errors.
1. Any Detection: 50+ tests scanning for `Any` usage outside explicit quarantine zones.

M-PY-S-Models

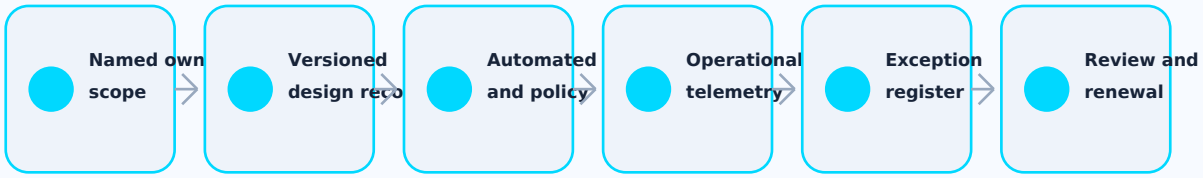
1. Dict Leak: 100+ tests scanning for `Dict[str, Any]` in function signatures.
1. Pydantic Strict: 50+ tests verifying `model_config = ConfigDict(strict=True)`.

Execution Protocol

ASSURANCE AND ADOPTION

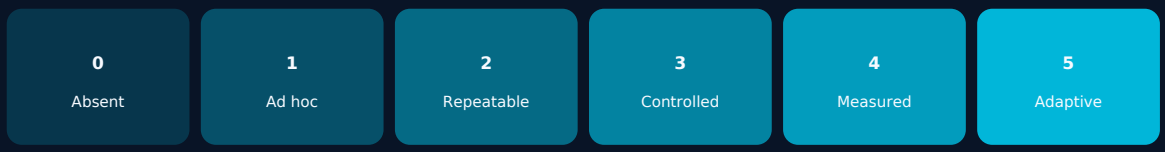
Implementation evidence and review gates

Required assurance chain



A passing score is not sufficient without current evidence. High-assurance adoption requires traceable design decisions, reproducible builds, independently reviewable tests, controlled secrets, runtime observability, failure exercises, and a dated exception register.

Assurance maturity spectrum



Current authority register

Authority	Version / snapshot	Applicability	Review by
Rust Project: The Rust Programming Language and Rust Reference	Rolling official documentation; book assumes Rust 1.90.0 or later and Rust 2024 Edition	High	2026-10-22
Go Project: The Go Programming Language Specification and Effective Go	Rolling official documentation snapshot	High	2026-10-22
Python Software Foundation: Python Language Reference, Packaging, Typing, and PEP Index	Rolling official documentation snapshot	High	2026-10-22
Tauri Programme: Tauri 2 Documentation	Tauri 2 rolling documentation snapshot	High	2026-10-22
Svelte Project: Svelte Documentation	Svelte 5 rolling documentation snapshot	High	2026-10-22
TLA+ Foundation: TLA+ Language and Tools Documentation	Rolling documentation snapshot	High	2027-01-24
OpenAPI Initiative: OpenAPI Specification	3.2.0	High	2027-01-24
OpenTelemetry Project: OpenTelemetry Documentation and Specification	Rolling specification snapshot; profiles public alpha and declarative configuration stable in 2026	High	2026-10-22

Research limitation. Official documentation establishes published interfaces and declared behavior. It does not prove the performance, security, interoperability, cost, or operational suitability of a specific implementation. Those claims require controlled testing and retained evidence.

ENGINEERING STANDARDS LIBRARY / SOFTWARE ENGINEERING

Polyglot Boundaries, Typed Contracts, and IPC Standard

Explicit language boundaries, schema-governed messages, compatibility policy, local IPC, remote RPC, and failure containment.

PERMANENT DOCUMENT ID

MYTHOS-SWE-0005

PUBLICATION

Candidate Standard
Version 1.0.0-candidate

ASSURANCE PROFILE

Standard / Advanced
High Assurance

PERMANENT PUBLICATION IDENTITY

Polyglot Boundaries, Typed Contracts, and IPC Standard

DOCUMENT ID
MYTHOS-SWE-0005

VERSION
1.0.0-candidate

STATUS
Candidate Standard

PUBLISHER
Mythos Systems

PUBLICATION DATE
2026-07-24

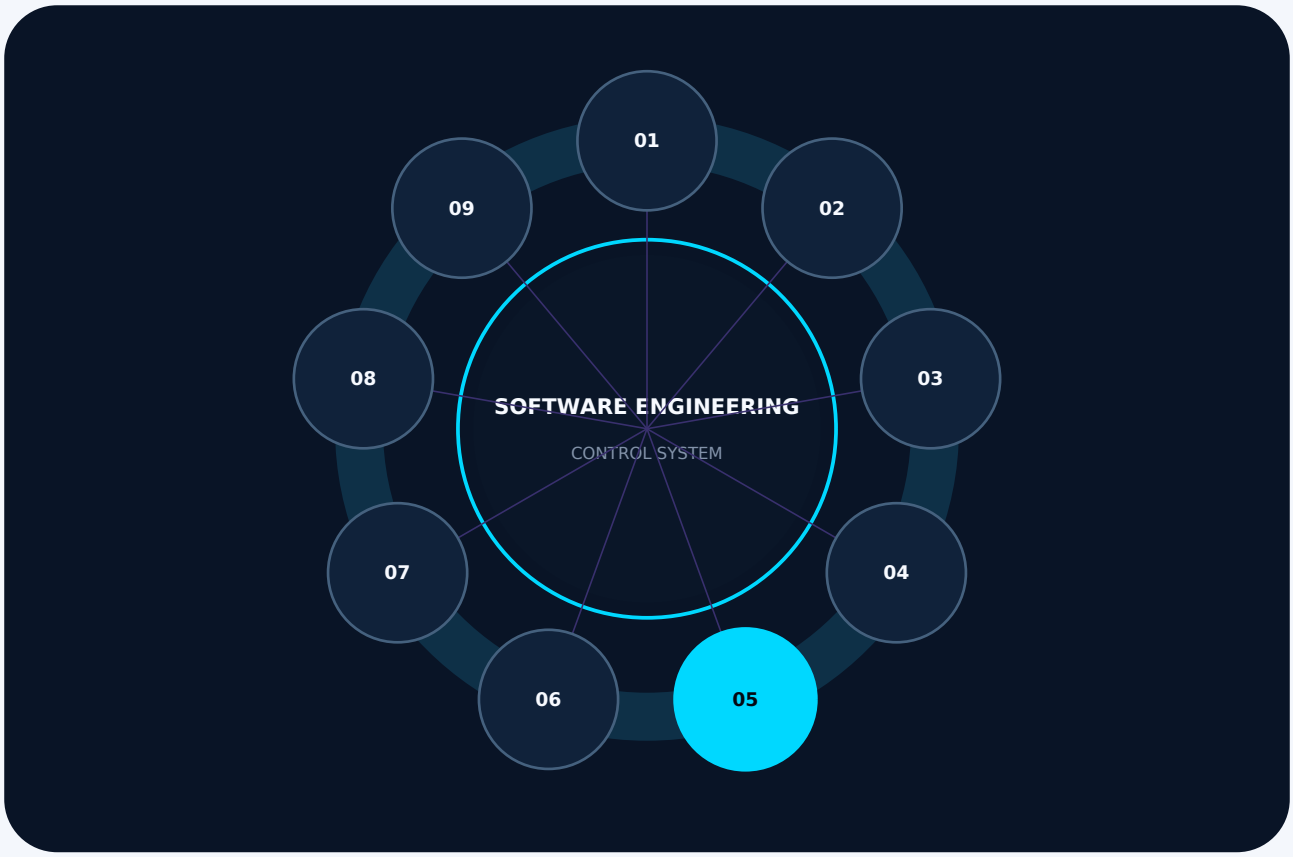
SCHEDULED REVIEW
2027-01-24

16	6	4	1
ATOMIC CRITERIA	ASSURANCE VIEWS	IMPLEMENTATION PROFILES	PERMANENT IDENTITY

Publication doctrine. This publication establishes durable guidance or requirements for its defined scope. Interpretation must preserve the permanent document identity, recorded evidence, controlled exceptions, and formal revision history.

DOMAIN CONTROL SYSTEM

MYTHOS-SWE-0005 inside the software engineering standard constellation



Connected assurance	Architecture, implementation, operations, evidence, and lifecycle are evaluated as one controlled system.
Specialization rule	Companion standards may add precision, but they cannot silently weaken this publication's requirements.

EXECUTIVE POSITION

Polyglot Boundaries, Typed Contracts, and IPC Standard

The integration of polyglot systems has failed.

Evidence-bearing delivery path

Assurance maturity spectrum

CONTENTS

Publication navigation

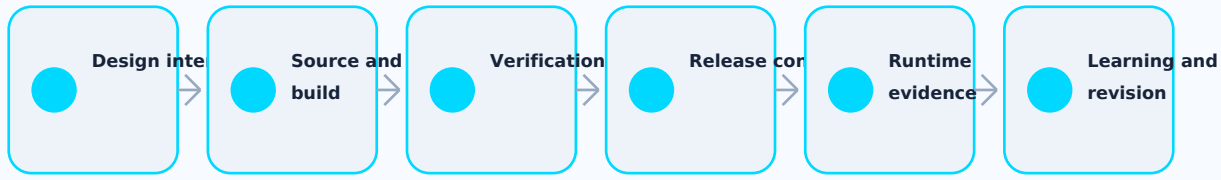
CONTROL TOPOLOGY

Assurance model and evaluation surface

The standard is assessed as an evidence-bearing system. Capability scores do not replace mandatory gates, implementation proof, or operational validation.

Capability family	Evaluation nodes
Typed Contracts and Schema-First Design (Weight: 25%)	2
Schema Evolution and Compatibility (Weight: 20%)	2
IPC Mechanisms and Performance (Weight: 20%)	2
Boundary Isolation and Shared DB Prohibition (Weight: 15%)	2
Contract Testing and Governance (Weight: 10%)	1
Polyglot Code Generation (Weight: 10%)	1
Suite Composition	2
Additional Evaluation Dimension	4

Lifecycle control loop



EVALUATION INDEX

Criteria and evidence map

#	Capability family	Criterion
1	Typed Contracts and Schema-First Design (Weight: 25%)	Schema Authority
2	Typed Contracts and Schema-First Design (Weight: 25%)	Type Strictness
3	Schema Evolution and Compatibility (Weight: 20%)	Breaking Change Prevention
4	Schema Evolution and Compatibility (Weight: 20%)	Versioning Strategy
5	IPC Mechanisms and Performance (Weight: 20%)	Internal Communication Protocol
6	IPC Mechanisms and Performance (Weight: 20%)	Latency and Overhead
7	Boundary Isolation and Shared DB Prohibition (Weight: 15%)	Shared Database Prohibition
8	Boundary Isolation and Shared DB Prohibition (Weight: 15%)	Anti-Corruption at Boundary
9	Contract Testing and Governance (Weight: 10%)	Consumer-Driven Contract Testing
10	Polyglot Code Generation (Weight: 10%)	Code Generation
11	Suite Composition	M-POLY-S-Contracts
12	Suite Composition	M-POLY-S-Isolation
13	Additional Evaluation Dimension	Boundary Dimensions
14	Additional Evaluation Dimension	The Boundary Doctrine
15	Additional Evaluation Dimension	Scoring Model
16	Additional Evaluation Dimension	Suite Composition

NORMATIVE PUBLICATION

Requirements, evaluation methodology, and implementation guidance

The following body preserves the substantive source standard while normalizing publication status, identity, navigation, and visual presentation.

Executive Mandate

The integration of polyglot systems has failed.

Organizations adopt multiple languages (Rust for performance, Go for control planes, Python for AI) to leverage their strengths, only to integrate them via the lowest common denominator: untyped JSON over HTTP REST. This results in runtime `KeyError` exceptions, implicit coupling via shared database tables, and versioning nightmares where adding a field to one service crashes three others.

This standard exists because:

1. Untyped JSON is a Liability: Passing `Dict[str, Any]` or `map[string]interface{}` across a network boundary is an architectural failure. If the contract is not machine-readable and strictly enforced, the integration will break at runtime.
2. A Shared Database is Not an API: Integrating services by reading another service's database tables couples the structural internals of both systems, making independent deployment impossible. Integration MUST occur via well-defined APIs, asynchronous events, or Anti-Corruption Layers.
3. REST is for Humans, gRPC is for Machines: RESTful JSON APIs are optimized for browser clients and human debugging. Internal service-to-service communication MUST use binary, strongly-typed, schema-first protocols (gRPC, Protobuf, Cap'n Proto) to ensure performance, determinism, and strict contracts.
4. Code-First is Anti-Contract: Generating an API specification from code annotations (e.g., Swagger/OpenAPI generated from Spring controllers) allows implementation details to leak into the contract. The Schema MUST be designed first; code is a derived artifact.

This document establishes the Mythos Polyglot Boundary Standard (MPBS), a comprehensive, evidence-based methodology for evaluating typed contracts, IPC mechanisms, and schema evolution in polyglot environments.

Scope and Applicability

In Scope

This standard covers the evaluation of:

1. Schema-First contract design (Protobuf, Smithy, OpenAPI)
1. Inter-Process Communication (IPC) protocols (gRPC, Cap'n Proto, Unix Domain Sockets)
1. Schema evolution and backward/forward compatibility

1. Polyglot code generation and type mapping
1. Shared database prohibition and API-only integration
1. Contract testing and compatibility verification

Out of Scope

1. General domain-driven design (covered in MYTHOS-SWE-0001)
1. Event-driven architecture and messaging (covered in MYTHOS-DAT-0004)
1. General network transport (covered in MYTHOS-NET-0006)

Audience

1. Principal engineers and software architects designing polyglot systems
1. Platform engineers building service meshes and API gateways
1. Security teams evaluating IPC attack surfaces
1. AI governance, risk, and compliance teams
1. Standards bodies seeking a reference polyglot methodology

Definitions and Taxonomy

Boundary Dimensions

Dimension	Definition
Schema-First	A design paradigm where the API contract (e.g., `.proto` file, `.smithy` model) is defined and reviewed before any implementation code is written.
Backward Compatibility	The ability of a service to process requests from older clients without breaking.
Forward Compatibility	The ability of an older client to process responses from a newer service without breaking (usually via unknown field ignorance).
Shared Database Anti-Pattern	The practice of integrating two services by allowing one to read/write directly to the other's database, bypassing the domain logic.
Contract Testing	Testing that verifies the contract (schema and behavior) between a consumer and a provider, ensuring no breaking changes are introduced.

The Boundary Doctrine

> Untyped JSON is a liability. A shared database is not an API. REST is for humans; gRPC is for machines. Code-first is anti-contract. If the schema is not the source of truth, the integration is a lie.

Evaluation Methodology

Scoring Model

Each capability is scored from 0 to 5.

Score	Meaning
0	Fails basic task; untyped JSON, shared DBs, no versioning
1	Basic OpenAPI but not enforced; REST only
2	Functional but lacks schema-first design or compatibility checks
3	Solid production performance; schema-first, gRPC, compatibility enforced
4	Strong performance; automated contract testing, zero shared DBs
5	Best-in-class; sets the standard for deterministic, typed polyglot systems

Weighting

Category	Weight	Rationale
Typed Contracts & Schema-First Design	25%	Untyped boundaries cause runtime crashes
Schema Evolution & Compatibility	20%	Breaking changes cause cascading outages
IPC Mechanisms & Performance	20%	JSON/REST is too slow and loose for internal comms
Boundary Isolation & Shared DB Prohibition	15%	Shared DBs destroy independent deployability
Contract Testing & Governance	10%	Unverified contracts drift
Polyglot Code Generation	10%	Manual serialization introduces bugs

Total: 100%

A system MUST score at least 3.0 in Typed Contracts and Boundary Isolation to be considered for production use.

Evaluation Categories

Typed Contracts and Schema-First Design (Weight: 25%)

Schema Authority

Is the API contract (schema) the source of truth, or is it derived from code?

Score	Criteria
0	No API specification; implementation is the contract
1	OpenAPI generated from code annotations (Code-First)
2	OpenAPI written manually but not enforced at runtime
3	Schema-First design (Protobuf, Smithy); code generated from schema
4	Schema-First + runtime enforcement (e.g., gRPC schema validation)
5	Perfect authority: schema is the immutable contract, formally verified, cryptographically signed, zero code-first generation

Type Strictness

Are all fields strictly typed (no `Any`, no `object`, no untyped JSON)?

Score	Criteria
0	`map` or `Dict[str, Any]` in contracts
1	Basic types but frequent use of `oneof` with untyped fallback
2	All fields typed; enums used for fixed values
3	Custom scalar types (e.g., `UUID`, `IPv4Address`, `Timestamp`)
4	Strict type mapping across all polyglot targets (Rust, Go, Python)
5	Perfect strictness: zero untyped fields, type-state modeling in schemas, formally verified type compatibility

Schema Evolution and Compatibility (Weight: 20%)

Breaking Change Prevention

Does the system prevent breaking changes from reaching production?

Score	Criteria
0	No compatibility checking; breaking changes deployed manually
1	Manual review of schema diffs
2	Automated diff tool (e.g., `buf breaking`) run locally
3	Automated compatibility check in CI; breaking changes block merge
4	Backward and forward compatibility guaranteed; unknown fields preserved
5	Perfect prevention: formally verified compatibility, automated migration tools, zero breaking changes possible

Versioning Strategy

How are API versions managed?

Score	Criteria
0	No versioning; URL path versioning (e.g., `/v1/`, `/v2/`)
1	Header-based versioning
2	Semantic versioning of the schema package
3	Continuous evolution without version bumps (Protobuf/Smithy style: add fields, never remove)
4	Schema registry with lifecycle management (deprecated -> sunset -> removed)
5	Perfect versioning: continuous evolution, formally verified sunset periods, automated client migration tracking

IPC Mechanisms and Performance (Weight: 20%)

Internal Communication Protocol

What protocol is used for service-to-service communication?

Score	Criteria
0	JSON over HTTP REST (internal)
1	JSON over HTTP with HTTP/2

Score	Criteria
2	gRPC (Protobuf over HTTP/2) for some services
3	gRPC mandatory for all internal service-to-service communication
4	gRPC + Unix Domain Sockets for sidecar/co-located communication
5	Perfect protocol: gRPC for distributed, Cap'n Proto/uDS for co-located, formally verified payload sizes, zero internal REST

Latency and Overhead

Is the IPC mechanism optimized for low latency and minimal serialization overhead?

Score	Criteria
0	High latency (JSON serialization/deserialization dominates)
1	gRPC with standard Protobuf serialization
2	Protobuf with zero-copy parsing (e.g., `bytes` fields)
3	gRPC with HTTP/2 multiplexing and connection pooling
4	Cap'n Proto or FlatBuffers for zero-copy inter-process communication
5	Perfect performance: zero-copy IPC, shared memory for co-located services, formally verified latency bounds

Boundary Isolation and Shared DB Prohibition (Weight: 15%)

Shared Database Prohibition

Are services prevented from accessing another service's database?

Score	Criteria
0	Services share databases freely
1	Read-only access to other services' databases
2	Logical separation but same physical database instance
3	Separate database instances; integration via API only
4	Network-level segregation; database credentials scoped to single service
5	Perfect isolation: cryptographic boundary enforcement, zero cross-service database access, formally verified data ownership

Anti-Corruption at Boundary

Are external models prevented from leaking into the domain?

Score	Criteria
0	External contract types (e.g., Protobuf messages) used in domain logic
1	Manual mapping between external and domain types
2	Automated mapping but in domain layer
3	Anti-Corruption Layer (ACL) at the boundary; domain uses its own types
4	ACL with generated mappers and contract tests

Score	Criteria
5	Perfect isolation: WASM-sandboxed ACL, formally verified translation, zero external schema leakage

Contract Testing and Governance (Weight: 10%)

Consumer-Driven Contract Testing

Are consumer expectations verified against the provider?

Score	Criteria
0	No contract testing; integration tested in staging only
1	Provider tests its own API
2	Consumer-driven contract tests (Pact) written but not in CI
3	Contract tests in CI for every PR
4	Contract tests + schema compatibility checks + backward compatibility verification
5	Perfect governance: formally verified consumer contracts, automated provider verification, zero untested integrations

Polyglot Code Generation (Weight: 10%)

Code Generation

Is client/server code generated from the schema, or written manually?

Score	Criteria
0	Manual HTTP clients and JSON parsing
1	Generated clients but manually maintained
2	Generated clients from schema (e.g., `buf generate`, `protoc`)
3	Generated clients in CI; schema update triggers client regeneration
4	Generated clients with strict type mapping and custom scalar support
5	Perfect generation: deterministic, reproducible, formally verified type mapping across Rust/Go/Python/TS, zero manual serialization

The Mythos Polyglot Suite (M-POLY-S)

Traditional benchmarks measure endpoint latency. The M-POLY-S evaluates contract strictness, compatibility, and boundary isolation.

Suite Composition

M-POLY-S-Contracts

1. Breaking Change: 100+ tests applying known breaking schema changes, verifying CI blocks them.
1. Compatibility: 50+ tests verifying old clients can communicate with new servers (forward compatibility).

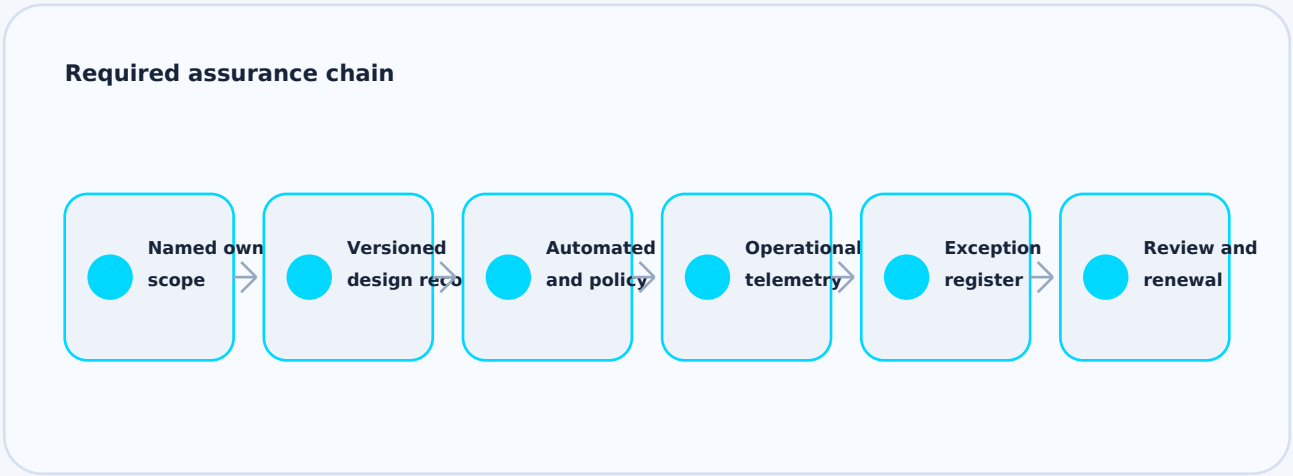
M-POLY-S-Isolation

1. Shared DB Scan: 50+ tests scanning for database credentials used by multiple services.
1. Untyped Field Scan: 100+ tests scanning for `Any`, `object`, or `map` in schemas.

Execution Protocol

ASSURANCE AND ADOPTION

Implementation evidence and review gates



A passing score is not sufficient without current evidence. High-assurance adoption requires traceable design decisions, reproducible builds, independently reviewable tests, controlled secrets, runtime observability, failure exercises, and a dated exception register.

Current authority register

Authority	Version / snapshot	Applicability	Review by
Rust Project: The Rust Programming Language and Rust Reference	Rolling official documentation; book assumes Rust 1.90.0 or later and Rust 2024 Edition	High	2026-10-22
Go Project: The Go Programming Language Specification and Effective Go	Rolling official documentation snapshot	High	2026-10-22
Python Software Foundation: Python Language Reference, Packaging, Typing, and PEP Index	Rolling official documentation snapshot	High	2026-10-22
Tauri Programme: Tauri 2 Documentation	Tauri 2 rolling documentation snapshot	High	2026-10-22
Svelte Project: Svelte Documentation	Svelte 5 rolling documentation snapshot	High	2026-10-22
TLA+ Foundation: TLA+ Language and Tools Documentation	Rolling documentation snapshot	High	2027-01-24
OpenAPI Initiative: OpenAPI Specification	3.2.0	High	2027-01-24
OpenTelemetry Project: OpenTelemetry Documentation and Specification	Rolling specification snapshot; profiles public alpha and declarative configuration stable in 2026	High	2026-10-22

Research limitation. Official documentation establishes published interfaces and declared behavior. It does not prove the performance, security, interoperability, cost, or operational suitability of a specific implementation. Those claims require controlled testing and retained evidence.

ENGINEERING STANDARDS LIBRARY / SOFTWARE ENGINEERING

Desktop Architecture and Tauri/Svelte Integration Standard

Secure desktop runtimes, Rust-native command boundaries, Svelte application state, capability control, and update integrity.

PERMANENT DOCUMENT ID

MYTHOS-SWE-0006

PUBLICATION

Candidate Standard
Version 1.0.0-candidate

ASSURANCE PROFILE

Standard / Advanced
High Assurance

PERMANENT PUBLICATION IDENTITY

Desktop Architecture and Tauri/Svelte Integration Standard

DOCUMENT ID
MYTHOS-SWE-0006

VERSION
1.0.0-candidate

STATUS
Candidate Standard

PUBLISHER
Mythos Systems

PUBLICATION DATE
2026-07-24

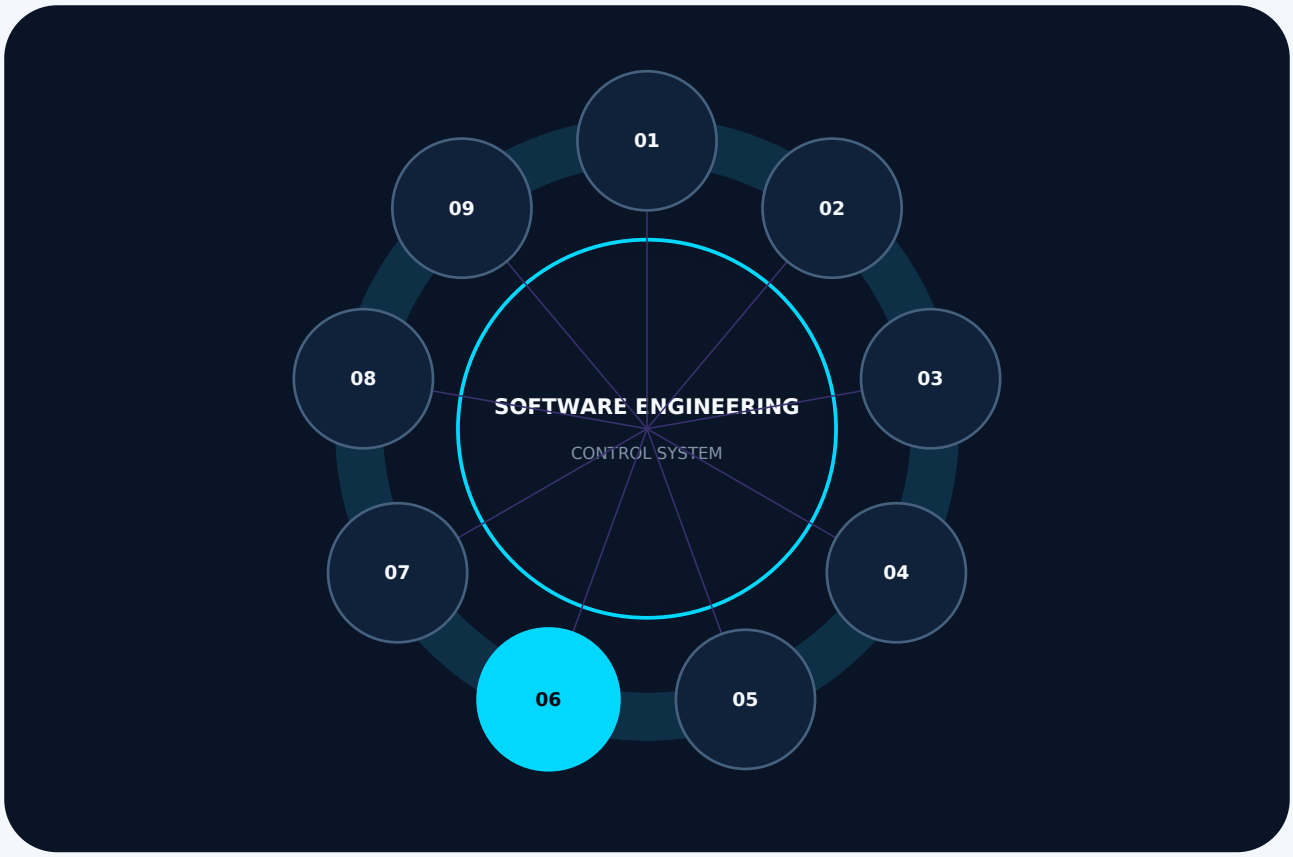
SCHEDULED REVIEW
2027-01-24

17	6	4	1
ATOMIC CRITERIA	ASSURANCE VIEWS	IMPLEMENTATION PROFILES	PERMANENT IDENTITY

Publication doctrine. This publication establishes durable guidance or requirements for its defined scope. Interpretation must preserve the permanent document identity, recorded evidence, controlled exceptions, and formal revision history.

DOMAIN CONTROL SYSTEM

MYTHOS-SWE-0006 inside the software engineering standard constellation



Connected assurance	Architecture, implementation, operations, evidence, and lifecycle are evaluated as one controlled system.
Specialization rule	Companion standards may add precision, but they cannot silently weaken this publication's requirements.

EXECUTIVE POSITION

Desktop Architecture and Tauri/Svelte Integration Standard

The architecture of desktop applications has failed.

Evidence-bearing delivery path

Assurance maturity spectrum

CONTENTS

Publication navigation

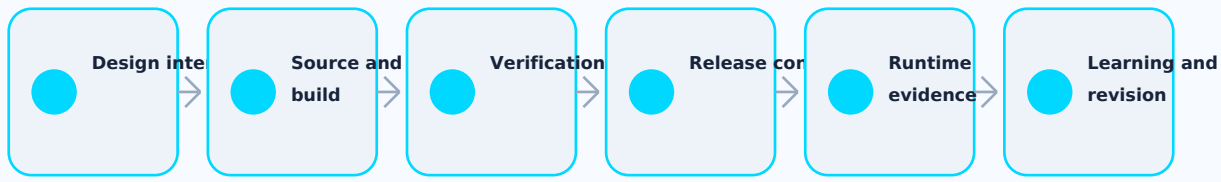
CONTROL TOPOLOGY

Assurance model and evaluation surface

The standard is assessed as an evidence-bearing system. Capability scores do not replace mandatory gates, implementation proof, or operational validation.

Capability family	Evaluation nodes
IPC Security and Renderer Isolation (Weight: 25%)	3
Backend Authority and Data Sovereignty (Weight: 20%)	2
Auto-Updater and Code Signing (Weight: 20%)	2
Frontend Discipline (Svelte) (Weight: 15%)	2
Local-First Data Architecture (Weight: 10%)	1
Performance and Resource Efficiency (Weight: 10%)	1
Suite Composition	2
Additional Evaluation Dimension	4

Lifecycle control loop



EVALUATION INDEX

Criteria and evidence map

#	Capability family	Criterion
1	IPC Security and Renderer Isolation (Weight: 25%)	IPC Validation
2	IPC Security and Renderer Isolation (Weight: 25%)	Tauri Allowlist Enforcement
3	IPC Security and Renderer Isolation (Weight: 25%)	Content Security Policy (CSP)
4	Backend Authority and Data Sovereignty (Weight: 20%)	Secret Isolation
5	Backend Authority and Data Sovereignty (Weight: 20%)	Business Logic Placement
6	Auto-Updater and Code Signing (Weight: 20%)	Update Signature Verification
7	Auto-Updater and Code Signing (Weight: 20%)	Atomic Updates and Rollback
8	Frontend Discipline (Svelte) (Weight: 15%)	Type Safety Across IPC
9	Frontend Discipline (Svelte) (Weight: 15%)	Dependency Minimalism (Frontend)
10	Local-First Data Architecture (Weight: 10%)	Local Storage
11	Performance and Resource Efficiency (Weight: 10%)	Resource Footprint
12	Suite Composition	M-DESK-S-IPC
13	Suite Composition	M-DESK-S-Update
14	Additional Evaluation Dimension	Desktop Dimensions
15	Additional Evaluation Dimension	The Desktop Doctrine
16	Additional Evaluation Dimension	Scoring Model
17	Additional Evaluation Dimension	Suite Composition

NORMATIVE PUBLICATION

Requirements, evaluation methodology, and implementation guidance

The following body preserves the substantive source standard while normalizing publication status, identity, navigation, and visual presentation.

Executive Mandate

The architecture of desktop applications has failed.

For a decade, the industry has defaulted to Electron, bundling an entire Chromium browser and a Node.js runtime into every desktop app. This results in 200MB "Hello World" applications, 1GB+ memory footprints, and a catastrophic attack surface where a single Cross-Site Scripting (XSS) vulnerability in a web view leads to Remote Code Execution (RCE) because the renderer has direct access to the filesystem and OS shell.

This standard exists because:

1. The Renderer is Untrusted: A web view (Chromium/WebView2) processes untrusted data from the internet, local files, and user inputs. Granting it system-level authority is an architectural failure. The renderer MUST be a thin, stateless projection of the backend.
2. Electron is a Liability: Bundling Node.js into the renderer bridges the untrusted web world with the trusted OS. Tauri solves this by using the OS-native WebView and a Rust backend, shrinking the attack surface by orders of magnitude.
3. IPC is a Network Boundary: The bridge between the Svelte frontend and the Rust backend must be treated with the same zero-trust rigor as a public API. All inputs from the renderer MUST be validated; the frontend is assumed compromised.
4. An Unsigned Update is a Backdoor: Auto-updaters that fetch and execute unsigned binaries are the easiest path to compromising a fleet. Updates MUST be cryptographically signed, ideally with PQC-ready algorithms, and verified before execution.

This document establishes the Mythos Desktop Architecture Standard (MDAS), a comprehensive, evidence-based methodology for evaluating Tauri/Svelte desktop applications against IPC security, authority separation, and update integrity.

Scope and Applicability

In Scope

This standard covers the evaluation of:

1. Desktop application architectures (Tauri, Electron, native)
1. Inter-Process Communication (IPC) between frontend and backend
1. WebView isolation and Content Security Policy (CSP)

1. Auto-updater security and code signing
1. Local-first data architecture (SQLite, `sqlite-vec`)
1. Svelte/SvelteKit integration patterns and discipline
1. Native OS integration (filesystem, shell, notifications)

Out of Scope

1. General Rust engineering (covered in MYTHOS-SWE-0002)
1. General web/browser engineering (reserved for MYTHOS-WEB)
1. General local-first sync architecture (covered in MYTHOS-DAT-0001)

Audience

1. Desktop application engineers and architects
1. Security teams evaluating desktop attack surfaces
1. Platform engineers building developer tooling
1. AI governance teams evaluating local agent UIs
1. Standards bodies seeking a reference desktop architecture methodology

---

Definitions and Taxonomy

Desktop Dimensions

Dimension	Definition
Tauri	A framework for building tiny, fast binaries for all major desktop platforms using the OS's native WebView and a Rust backend.
IPC (Inter-Process Communication)	The protocol bridge (e.g., Tauri's `invoke`) allowing the untrusted WebView (Svelte) to request actions from the trusted Rust backend.
Renderer Isolation	The architectural principle that the WebView (renderer) has zero direct access to the OS, filesystem, or network; all access is brokered by the Rust backend.
CSP	Content Security Policy. A browser security standard that restricts what resources the WebView is allowed to load (e.g., no inline scripts, no external domains).

The Desktop Doctrine

> The renderer is untrusted. The backend is the authority. IPC is a network boundary. An unsigned update is a backdoor. A 200MB hello-world is a failure.

Evaluation Methodology

Scoring Model

Each capability is scored from 0 to 5.

Score	Meaning
0	Fails basic task; Electron, Node in renderer, unvalidated IPC
1	Basic Tauri but overly permissive allowlists or no CSP
2	Functional but lacks strict IPC validation or update signing
3	Solid production performance; strict CSP, validated IPC, signed updates
4	Strong performance; PQC updates, local-first encrypted data, typed IPC
5	Best-in-class; sets the standard for zero-trust desktop architecture

Weighting

Category	Weight	Rationale
IPC Security & Renderer Isolation	25%	XSS to RCE is the #1 desktop threat
Backend Authority & Data Sovereignty	20%	The frontend must not hold secrets or logic
Auto-Updater & Code Signing	20%	A compromised updater owns the fleet
Frontend Discipline (Svelte)	15%	Logic in the frontend is unverified logic
Local-First Data Architecture	10%	Cloud-dependent desktops are fragile
Performance & Resource Efficiency	10%	Resource waste is an architectural failure

Total: 100%

A system MUST score at least 3.0 in IPC Security and Auto-Updater to be considered for production use.

Evaluation Categories

IPC Security and Renderer Isolation (Weight: 25%)

IPC Validation

Are all inputs from the frontend validated and sanitized in the Rust backend?

Score	Criteria
0	No validation; frontend data trusted implicitly
1	Basic type checking but no domain validation
2	Strict type validation using Serde/Schemars for all IPC commands
3	Domain validation (e.g., path traversal prevention, command allowlisting)
4	Sandboxed IPC: frontend cannot invoke commands not explicitly registered and typed
5	Perfect security: formally validated IPC schemas, zero implicit trust, WASM-level sandboxing of IPC handlers

Tauri Allowlist Enforcement

Is the Tauri allowlist restricted to the absolute minimum required capabilities?

Score	Criteria
0	`allowlist: { all: true }` or equivalent
1	Broad permissions (e.g., `fs: { scope: ["**"] }`)
2	Scope restricted to specific directories
3	Scope restricted to specific file patterns; shell disabled
4	Granular capability-based permissions (Tauri v2 capabilities)
5	Perfect enforcement: zero wildcard scopes, formally verified minimal surface, dynamic capability grants

Content Security Policy (CSP)

Is a strict CSP enforced on the WebView?

Score	Criteria
0	No CSP; external scripts loadable
1	Basic CSP but allows `unsafe-inline` or `unsafe-eval`
2	Strict CSP; no inline scripts; no eval
3	Nonce-based or hash-based CSP for all inline needs
4	CSP restricts connections to specific backend origins; no external CDNs
5	Perfect CSP: hash-based, zero external connections, formally verified policy, no `unsafe-*`

Backend Authority and Data Sovereignty (Weight: 20%)

Secret Isolation

Are secrets (API keys, tokens, passwords) prevented from entering the frontend?

Score	Criteria
0	Secrets stored in `localStorage` or JS variables
1	Secrets fetched by frontend and passed to backend
2	Secrets stored by backend; references sent to frontend
3	Just-in-time secret brokering; frontend never sees raw secret
4	Hardware-backed key storage (OS Keychain/DPAPI) managed by Rust
5	Perfect isolation: zero secrets in WebView memory, cryptographic attestation of secret usage, PQC-encrypted local vault

Business Logic Placement

Is business logic restricted to the Rust backend?

Score	Criteria
0	Complex business logic in Svelte stores
1	Logic split evenly between frontend and backend

Score	Criteria
2	Logic primarily in backend; frontend handles routing/display
3	Frontend is a "thin client"; only UI state and rendering
4	Typed state synchronization; frontend state is a projection of backend state
5	Perfect placement: frontend is a pure function of backend state, formally verified authority boundaries

Auto-Updater and Code Signing (Weight: 20%)

Update Signature Verification

Are updates cryptographically signed and verified before execution?

Score	Criteria
0	No auto-update or unsigned updates
1	HTTPS-only verification (relies on PKI)
2	Ed25519 signature verification on update binary
3	Signature + Transparency log (Sigstore/Rekor) verification
4	PQC signature support (ML-DSA) for future-proofing
5	Perfect signing: PQC/hybrid signatures, transparency log, hardware-rooted verification, zero unsigned execution

Atomic Updates and Rollback

Are updates atomic and instantly rollback-safe?

Score	Criteria
0	In-place mutation; failed update corrupts install
1	Backup created but manual rollback required
2	Dual-directory layout (current/next); fallback on crash
3	Atomic swap with automatic rollback on startup failure
4	Delta updates with verified patching
5	Perfect updates: atomic, delta, PQC-signed, formally verified rollback, zero user disruption

Frontend Discipline (Svelte) (Weight: 15%)

Type Safety Across IPC

Are types consistent between Rust backend and Svelte frontend?

Score	Criteria
0	Manual type definition; TypeScript and Rust types drift
1	TypeScript types manually synced with Rust Serde schemas
2	TypeScript types auto-generated from Rust (e.g., `ts-rs`, `specta`)
3	Generated types + runtime validation (Zod) on IPC responses

Score	Criteria
4	End-to-end type safety: Rust -> TypeScript -> Svelte stores
5	Perfect discipline: auto-generated, runtime-validated, formally verified schema compatibility

Dependency Minimalism (Frontend)

Is the frontend dependency tree minimal and audited?

Score	Criteria
0	Heavy framework (Next.js on desktop); massive node_modules
1	SvelteKit but with many third-party components
2	Minimal Svelte; no heavy UI libraries
3	No external runtime dependencies; only dev-dependencies
4	Sub-resource integrity for any loaded assets
5	Perfect minimalism: zero runtime npm dependencies, audited dev-dependencies, strictly typed

Local-First Data Architecture (Weight: 10%)

Local Storage

Is user data stored locally-first, encrypted, and sovereign?

Score	Criteria
0	Cloud-only storage; no offline capability
1	Local SQLite but unencrypted
2	SQLite with SQLCipher encryption at rest
3	Local-first sync architecture (`sqlite-vec` for vectors)
4	CRDT-based synchronization for multi-device
5	Perfect local-first: encrypted SQLite, `sqlite-vec`, CRDT sync, zero data loss, formally verified consistency

Performance and Resource Efficiency (Weight: 10%)

Resource Footprint

Does the application respect system resources (RAM, CPU, binary size)?

Score	Criteria
0	>500MB RAM idle; >200MB binary; >5s startup (Electron baseline)
1	<200MB RAM idle; <100MB binary; <3s startup
2	<100MB RAM idle; <50MB binary; <2s startup
3	<50MB RAM idle; <20MB binary; <1s startup
4	<20MB RAM idle; <10MB binary; <500ms startup
5	Perfect efficiency: <10MB RAM idle, <5MB binary, <200ms startup, formally verified resource bounds

The Mythos Desktop Suite (M-DESK-S)

Traditional desktop benchmarks measure feature completeness. The M-DESK-S evaluates IPC isolation, update security, and resource discipline.

Suite Composition

M-DESK-S-IPC

1. XSS to RCE: 100+ tests injecting malicious payloads into WebView inputs to verify IPC validation prevents execution.
1. Allowlist Violation: 50+ tests attempting filesystem/network access from unregistered IPC commands.

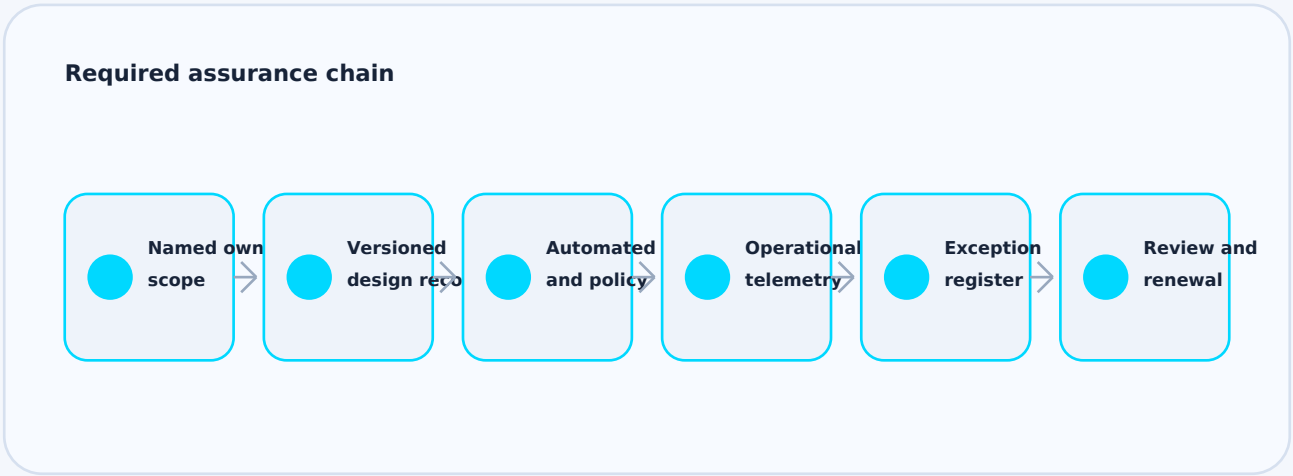
M-DESK-S-Update

1. Malicious Update: 50+ tests serving tampered binaries to verify the updater rejects them.
1. Rollback: 50+ tests interrupting updates to verify atomic rollback.

Execution Protocol

ASSURANCE AND ADOPTION

Implementation evidence and review gates



A passing score is not sufficient without current evidence. High-assurance adoption requires traceable design decisions, reproducible builds, independently reviewable tests, controlled secrets, runtime observability, failure exercises, and a dated exception register.

Current authority register

Authority	Version / snapshot	Applicability	Review by
Rust Project: The Rust Programming Language and Rust Reference	Rolling official documentation; book assumes Rust 1.90.0 or later and Rust 2024 Edition	High	2026-10-22
Go Project: The Go Programming Language Specification and Effective Go	Rolling official documentation snapshot	High	2026-10-22
Python Software Foundation: Python Language Reference, Packaging, Typing, and PEP Index	Rolling official documentation snapshot	High	2026-10-22
Tauri Programme: Tauri 2 Documentation	Tauri 2 rolling documentation snapshot	High	2026-10-22
Svelte Project: Svelte Documentation	Svelte 5 rolling documentation snapshot	High	2026-10-22
TLA+ Foundation: TLA+ Language and Tools Documentation	Rolling documentation snapshot	High	2027-01-24
OpenAPI Initiative: OpenAPI Specification	3.2.0	High	2027-01-24
OpenTelemetry Project: OpenTelemetry Documentation and Specification	Rolling specification snapshot; profiles public alpha and declarative configuration stable in 2026	High	2026-10-22

Research limitation. Official documentation establishes published interfaces and declared behavior. It does not prove the performance, security, interoperability, cost, or operational suitability of a specific implementation. Those claims require controlled testing and retained evidence.

ENGINEERING STANDARDS LIBRARY / SOFTWARE ENGINEERING

Formal Verification and Deterministic State Machine Standard

Executable specifications, invariant verification, deterministic transitions, model checking, and traceable implementation refinement.

PERMANENT DOCUMENT ID

MYTHOS-SWE-0007

PUBLICATION

Candidate Standard
Version 1.0.0-candidate

ASSURANCE PROFILE

Standard / Advanced
High Assurance

PERMANENT PUBLICATION IDENTITY

Formal Verification and Deterministic State Machine Standard

DOCUMENT ID
MYTHOS-SWE-0007

VERSION
1.0.0-candidate

STATUS
Candidate Standard

PUBLISHER
Mythos Systems

PUBLICATION DATE
2026-07-24

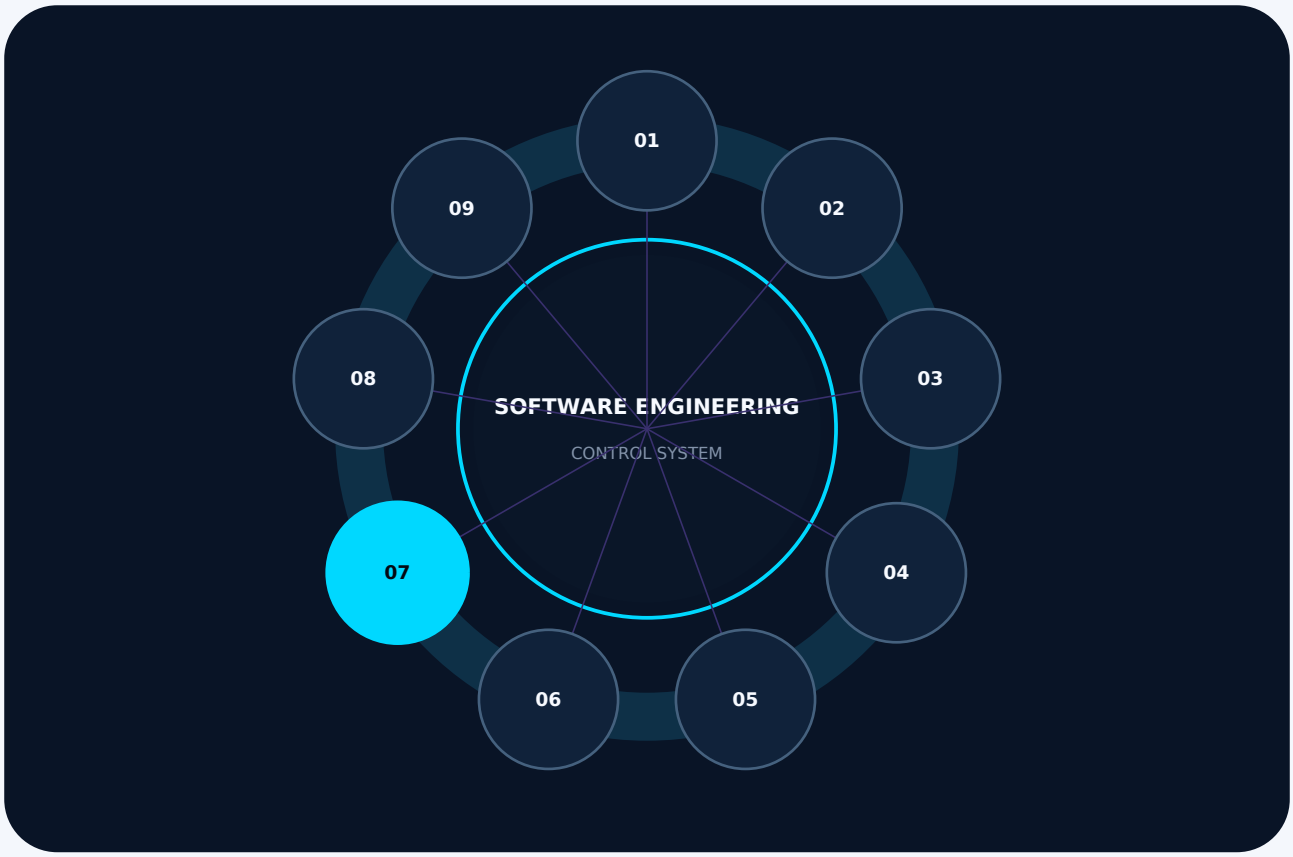
SCHEDULED REVIEW
2027-01-24

16	6	4	1
ATOMIC CRITERIA	ASSURANCE VIEWS	IMPLEMENTATION PROFILES	PERMANENT IDENTITY

Publication doctrine. This publication establishes durable guidance or requirements for its defined scope. Interpretation must preserve the permanent document identity, recorded evidence, controlled exceptions, and formal revision history.

DOMAIN CONTROL SYSTEM

MYTHOS-SWE-0007 inside the software engineering standard constellation



Connected assurance	Architecture, implementation, operations, evidence, and lifecycle are evaluated as one controlled system.
Specialization rule	Companion standards may add precision, but they cannot silently weaken this publication's requirements.

EXECUTIVE POSITION

Formal Verification and Deterministic State Machine Standard

The validation of critical state machines has failed.

Evidence-bearing delivery path

Assurance maturity spectrum

CONTENTS

Publication navigation

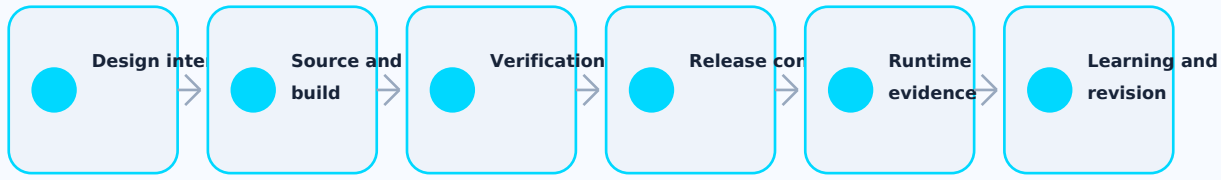
CONTROL TOPOLOGY

Assurance model and evaluation surface

The standard is assessed as an evidence-bearing system. Capability scores do not replace mandatory gates, implementation proof, or operational validation.

Capability family	Evaluation nodes
Safety and Invariant Verification (Weight: 25%)	1
Liveness and Temporal Logic (Weight: 20%)	2
Specification-to-Code Alignment (Weight: 20%)	2
State Machine Architecture (Weight: 15%)	2
Tooling and CI Integration (Weight: 10%)	1
Operational Lifecycle (Weight: 10%)	1
Suite Composition	2
Additional Evaluation Dimension	5

Lifecycle control loop



EVALUATION INDEX

Criteria and evidence map

#	Capability family	Criterion
1	Safety and Invariant Verification (Weight: 25%)	Deadlock Freedom
2	Liveness and Temporal Logic (Weight: 20%)	Termination Guarantees
3	Liveness and Temporal Logic (Weight: 20%)	Fairness and Starvation
4	Specification-to-Code Alignment (Weight: 20%)	Refinement Mapping
5	Specification-to-Code Alignment (Weight: 20%)	Conformance Testing
6	State Machine Architecture (Weight: 15%)	Determinism
7	State Machine Architecture (Weight: 15%)	State Space Bounding
8	Tooling and CI Integration (Weight: 10%)	Specification in CI
9	Operational Lifecycle (Weight: 10%)	Specification Maintenance
10	Suite Composition	MFVS-Safety
11	Suite Composition	MFVS-Liveness
12	Additional Evaluation Dimension	Verification Dimensions
13	Additional Evaluation Dimension	The Verification Doctrine
14	Additional Evaluation Dimension	Scoring Model
15	Additional Evaluation Dimension	Suite Composition
16	Additional Evaluation Dimension	Execution Protocol

NORMATIVE PUBLICATION

Requirements, evaluation methodology, and implementation guidance

The following body preserves the substantive source standard while normalizing publication status, identity, navigation, and visual presentation.

Executive Mandate

The validation of critical state machines has failed.

Organizations rely on unit tests, integration tests, and property-based testing to verify the correctness of distributed systems and agentic execution planes. Testing, however, is sampling. It proves the presence of bugs, not their absence. For critical state machines—where a deadlock stalls a million-dollar GPU training job, or a livelock drains a cloud budget—testing is fundamentally insufficient.

This standard exists because:

1. Testing is Not Proof: A test suite verifies a specific execution path. Formal verification (TLA+, Apalache) proves mathematically that *no* execution path violates an invariant.
2. Concurrency Bugs are Invisible to Tests: Race conditions, deadlocks, and livelocks are Heisenbugs that rarely reproduce in test environments. Formal verification exhaustively explores the state space, proving deadlock freedom.
3. State Machines are the Core of Agents: An AI agent is a state machine. If the state machine allows invalid transitions (e.g., "Executing" to "Completed" without "Verifying"), the agent will eventually take that invalid path in production.
4. Determinism is Not Optional: In governed AI and infrastructure, the same inputs MUST produce the same state transitions. Non-deterministic state machines are untestable, unverifiable, and ungovernable.

This document establishes the Mythos Formal Verification Standard (MFVS), a comprehensive, evidence-based methodology for evaluating the use of formal specification, model checking, and deterministic state machine design in production systems.

Scope and Applicability

In Scope

This standard covers the evaluation of:

1. Formal specification languages (TLA+, PlusCal, Alloy)
1. Model checkers (TLC, Apalache, NuSMV)
1. Safety properties (invariants, deadlock freedom, state constraints)

1. Liveness properties (termination, fairness, eventual consistency)
1. Deterministic state machine design (finite state machines, hierarchical state machines)
1. Refinement mapping (specification to implementation alignment)
1. State space optimization and bounded model checking

Out of Scope

1. General software design principles (covered in MYTHOS-SWE-0001)
1. Specific language engineering (covered in MYTHOS-SWE-0002 through 0004)
1. General testing methodologies (unit, integration, property-based)

Audience

1. Principal engineers and architects designing critical state machines
1. Security teams evaluating invariant enforcement
1. Platform engineers building agentic execution planes
1. AI governance, risk, and compliance teams
1. Standards bodies seeking a reference formal verification methodology

Definitions and Taxonomy

Verification Dimensions

Dimension	Definition
Safety Property	A property that asserts "nothing bad ever happens." E.g., the system never enters an invalid state, two agents never hold the same exclusive lock.
Liveness Property	A property that asserts "something good eventually happens." E.g., the system eventually terminates, a requested resource is eventually granted.
Invariant	A predicate that must be true in every reachable state of the system.
Model Checking	An automated technique for verifying finite-state concurrent systems by exhaustively exploring the state space.
Refinement Mapping	A formal proof that an implementation (low-level specification) correctly implements an abstract specification (high-level specification).
State Explosion	The exponential growth of the state space that makes exhaustive model checking computationally intractable.

The Verification Doctrine

> Testing shows the presence of bugs. Formal verification proves their absence. A deadlock is an outage. A livelock is a budget fire. If it is critical, it must be proven.

Evaluation Methodology

Scoring Model

Each capability is scored from 0 to 5.

Score	Meaning
0	Fails basic task; no formal verification, ad-hoc state machines
1	Basic state diagrams but no formal specification
2	Functional TLA+ specs but not checked in CI or unmaintained
3	Solid production performance; TLA+/Apalache in CI, safety properties proven
4	Strong performance; liveness proven, refinement mapping, bounded state
5	Best-in-class; sets the standard for mathematically proven systems

Weighting

Category	Weight	Rationale
Safety & Invariant Verification	25%	Invalid states cause outages and breaches
Liveness & Temporal Logic	20%	Livelocks and starvation cause resource exhaustion
Specification-to-Code Alignment	20%	An unimplemented spec is a fiction
State Machine Architecture	15%	Non-determinism makes verification impossible
Tooling & CI Integration	10%	Specs must be continuously verified
Operational Lifecycle	10%	Specs must evolve with the system

Total: 100%

A system MUST score at least 3.0 in Safety and Specification-to-Code Alignment to be considered for production use in critical paths.

Evaluation Categories

Safety and Invariant Verification (Weight: 25%)

Invariant Definition

Are critical safety properties formally defined and proven?

Score	Criteria
0	No formal invariants; safety assumed via testing
1	Invariants documented in natural language
2	Invariants defined in TLA+ but not model checked
3	Invariants model checked (TLC/Apalache) for bounded state space
4	Invariants proven for unbounded state space using inductive invariants

Score	Criteria
5	Perfect safety: formally proven inductive invariants, zero invariant violations possible, mathematical proof of deadlock freedom

Deadlock Freedom

Is the system mathematically proven to be deadlock-free?

Score	Criteria
0	Deadlocks discovered in production
1	Deadlocks tested via chaos engineering
2	TLA+ model checked for deadlocks on small state space
3	Apalache bounded model checking for deadlocks
4	Proof of deadlock freedom via inductive invariant
5	Perfect proof: formally verified deadlock freedom, zero circular waits, mathematically proven resource allocation

---

Liveness and Temporal Logic (Weight: 20%)

Termination Guarantees

Can the system prove that critical operations eventually terminate?

Score	Criteria
0	No termination guarantees; processes hang indefinitely
1	Timeouts used as a proxy for liveness
2	Liveness properties defined in TLA+ (e.g., `<>[]Terminated`)
3	Liveness properties model checked with fairness assumptions
4	Proven termination under weak fairness
5	Perfect liveness: formally proven termination, zero starvation, proven progress under adversarial conditions

Fairness and Starvation

Does the system guarantee that all processes make progress?

Score	Criteria
0	No fairness guarantees; starvation observed
1	Round-robin scheduling in implementation (not proven)
2	Strong fairness assumed but not proven
3	Weak fairness proven in specification
4	Strong fairness proven for critical resources
5	Perfect fairness: formally proven no starvation, equitable resource distribution mathematically guaranteed

---

Specification-to-Code Alignment (Weight: 20%)

Refinement Mapping

Is there a formal mapping between the abstract specification and the implementation?

Score	Criteria
0	Specification and implementation unrelated
1	Specification written after implementation as documentation
2	Specification drives design, but no formal mapping
3	Informal refinement mapping documented
4	Formal refinement mapping defined; implementation proven to refine specification
5	Perfect alignment: formally verified refinement, automated conformance testing, zero specification-implementation drift

Conformance Testing

Is the implementation continuously tested against the specification?

Score	Criteria
0	No conformance testing
1	Manual verification during code review
2	Property-based tests derived from specification
3	Model-based testing generating tests from TLA+ specs
4	Runtime monitoring verifying implementation matches spec
5	Perfect conformance: continuous runtime verification, automated trace validation, zero unobservable state deviations

State Machine Architecture (Weight: 15%)

Determinism

Are state machines strictly deterministic?

Score	Criteria
0	Non-deterministic state transitions; behavior depends on timing
1	Mostly deterministic but with implicit non-determinism (e.g., hash map iteration)
2	Explicit determinism in design but not enforced
3	Deterministic state transitions enforced by type system (e.g., Rust type-state)
4	Formally verified determinism in specification
5	Perfect determinism: mathematically proven deterministic transitions, zero non-deterministic behavior, reproducible execution

State Space Bounding

Is the state space bounded and manageable?

Score	Criteria
0	Unbounded state space; model checking impossible
1	State space bounded by constants but too large for exhaustive checking
2	State space abstracted for model checking
3	Symmetry reduction and abstraction techniques applied
4	Bounded model checking (Apalache) handles realistic state space
5	Perfect bounding: state space fully explored, formally verified abstraction, zero state explosion

Tooling and CI Integration (Weight: 10%)

Specification in CI

Is formal verification a mandatory gate in the CI/CD pipeline?

Score	Criteria
0	Specifications run locally, if at all
1	Specifications run on commit, but failures are warnings
2	Specifications run in CI; failures block merge
3	Apalache bounded model checking in CI; fast feedback loop
4	Incremental verification; only changed modules re-verified
5	Perfect integration: formally verified gates, cryptographic signing of verification results, zero unverified merges

Operational Lifecycle (Weight: 10%)

Specification Maintenance

Are specifications maintained with the same rigor as code?

Score	Criteria
0	Specifications abandoned after initial design
1	Specifications updated manually, sporadically
2	Specifications version-controlled alongside code
3	Specification changes required for any state machine modification
4	Automated drift detection between spec and implementation
5	Perfect maintenance: formally verified spec-code parity, automated consistency checks, zero specification rot

The Mythos Formal Verification Suite (MFVS)

Traditional benchmarks measure code coverage. The MFVS evaluates mathematical proof, liveness, and specification alignment.

Suite Composition

MFVS-Safety

1. Invariant Violation: 100+ tests injecting adversarial states into the model checker.
1. Deadlock Injection: 50+ tests simulating resource contention to verify deadlock freedom proofs.

MFVS-Liveness

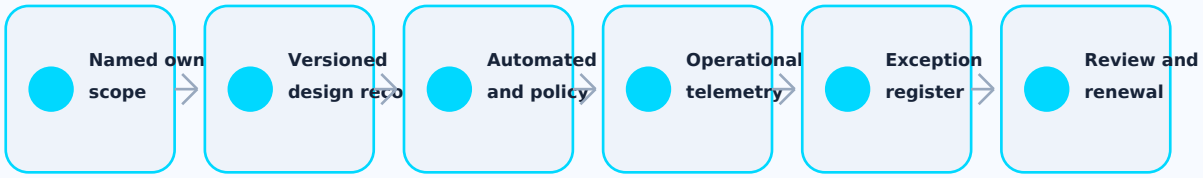
1. Starvation Simulation: 50+ tests simulating high-load to verify fairness properties.
1. Termination: 50+ tests verifying all operations reach a terminal state.

Execution Protocol

ASSURANCE AND ADOPTION

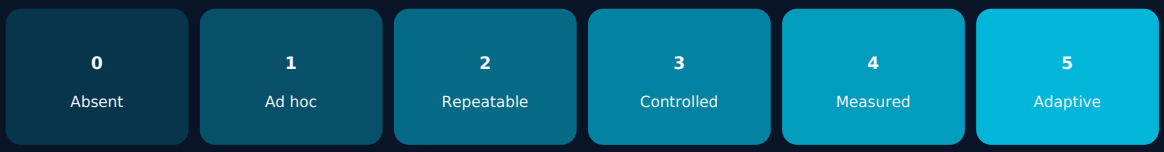
Implementation evidence and review gates

Required assurance chain



A passing score is not sufficient without current evidence. High-assurance adoption requires traceable design decisions, reproducible builds, independently reviewable tests, controlled secrets, runtime observability, failure exercises, and a dated exception register.

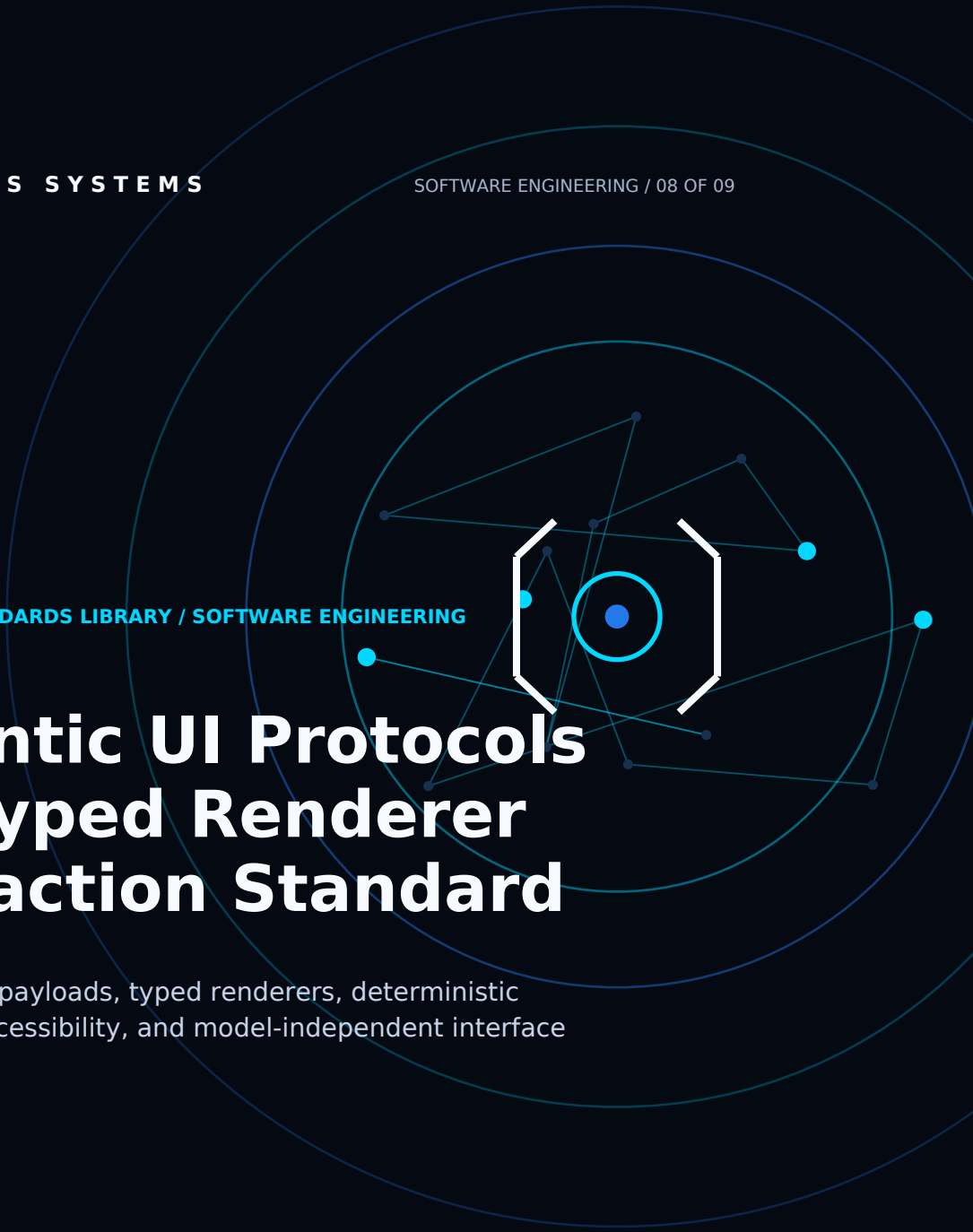
Assurance maturity spectrum



Current authority register

Authority	Version / snapshot	Applicability	Review by
Rust Project: The Rust Programming Language and Rust Reference	Rolling official documentation; book assumes Rust 1.90.0 or later and Rust 2024 Edition	High	2026-10-22
Go Project: The Go Programming Language Specification and Effective Go	Rolling official documentation snapshot	High	2026-10-22
Python Software Foundation: Python Language Reference, Packaging, Typing, and PEP Index	Rolling official documentation snapshot	High	2026-10-22
Tauri Programme: Tauri 2 Documentation	Tauri 2 rolling documentation snapshot	High	2026-10-22
Svelte Project: Svelte Documentation	Svelte 5 rolling documentation snapshot	High	2026-10-22
TLA+ Foundation: TLA+ Language and Tools Documentation	Rolling documentation snapshot	High	2027-01-24
OpenAPI Initiative: OpenAPI Specification	3.2.0	High	2027-01-24
OpenTelemetry Project: OpenTelemetry Documentation and Specification	Rolling specification snapshot; profiles public alpha and declarative configuration stable in 2026	High	2026-10-22

Research limitation. Official documentation establishes published interfaces and declared behavior. It does not prove the performance, security, interoperability, cost, or operational suitability of a specific implementation. Those claims require controlled testing and retained evidence.



ENGINEERING STANDARDS LIBRARY / SOFTWARE ENGINEERING

Semantic UI Protocols and Typed Renderer Abstraction Standard

Semantic intent payloads, typed renderers, deterministic presentation, accessibility, and model-independent interface contracts.

PERMANENT DOCUMENT ID

MYTHOS-SWE-0008

PUBLICATION

Candidate Standard
Version 1.0.0-candidate

ASSURANCE PROFILE

Standard / Advanced
High Assurance

PERMANENT PUBLICATION IDENTITY

Semantic UI Protocols and Typed Renderer Abstraction Standard

DOCUMENT ID
MYTHOS-SWE-0008

VERSION
1.0.0-candidate

STATUS
Candidate Standard

PUBLISHER
Mythos Systems

PUBLICATION DATE
2026-07-24

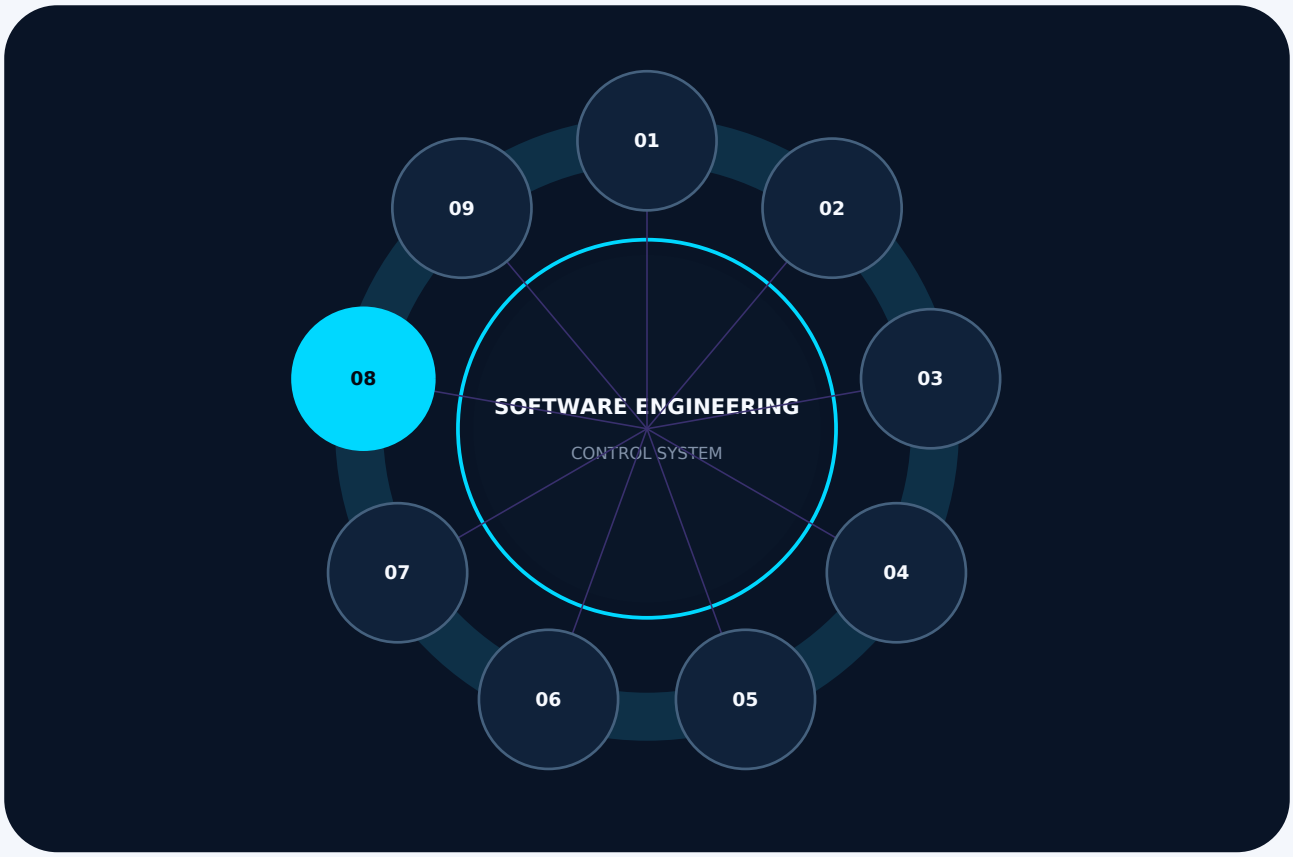
SCHEDULED REVIEW
2027-01-24

16	6	4	1
ATOMIC CRITERIA	ASSURANCE VIEWS	IMPLEMENTATION PROFILES	PERMANENT IDENTITY

Publication doctrine. This publication establishes durable guidance or requirements for its defined scope. Interpretation must preserve the permanent document identity, recorded evidence, controlled exceptions, and formal revision history.

DOMAIN CONTROL SYSTEM

MYTHOS-SWE-0008 inside the software engineering standard constellation



Connected assurance	Architecture, implementation, operations, evidence, and lifecycle are evaluated as one controlled system.
Specialization rule	Companion standards may add precision, but they cannot silently weaken this publication's requirements.

EXECUTIVE POSITION

Semantic UI Protocols and Typed Renderer Abstraction Standard

The integration of Large Language Models with user interfaces has failed.

Evidence-bearing delivery path

Assurance maturity spectrum

CONTENTS

Publication navigation

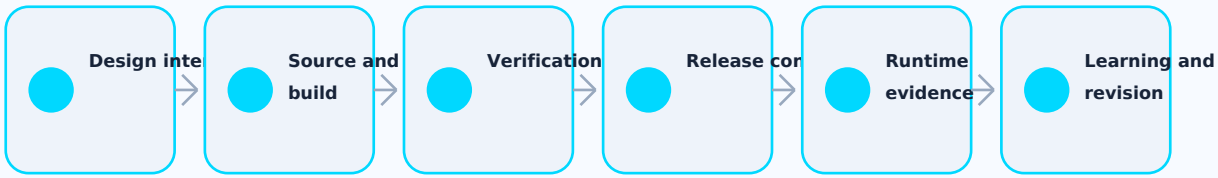
CONTROL TOPOLOGY

Assurance model and evaluation surface

The standard is assessed as an evidence-bearing system. Capability scores do not replace mandatory gates, implementation proof, or operational validation.

Capability family	Evaluation nodes
Semantic Payload Contracts (Weight: 25%)	2
Renderer Abstraction and Multi-Target (Weight: 20%)	2
Security and Injection Prevention (Weight: 20%)	2
High-Density and Accelerated Rendering (Weight: 15%)	2
State and Lifecycle Management (Weight: 10%)	1
Operational Lifecycle (Weight: 10%)	1
Suite Composition	2
Additional Evaluation Dimension	4

Lifecycle control loop



EVALUATION INDEX

Criteria and evidence map

#	Capability family	Criterion
1	Semantic Payload Contracts (Weight: 25%)	Intent vs. Markup
2	Semantic Payload Contracts (Weight: 25%)	Payload Schema Governance
3	Renderer Abstraction and Multi-Target (Weight: 20%)	Renderer Trait
4	Renderer Abstraction and Multi-Target (Weight: 20%)	Cross-Medium Fidelity
5	Security and Injection Prevention (Weight: 20%)	Markup Injection Prevention
6	Security and Injection Prevention (Weight: 20%)	Payload Validation
7	High-Density and Accelerated Rendering (Weight: 15%)	Accelerated Graphics
8	High-Density and Accelerated Rendering (Weight: 15%)	Data Density
9	State and Lifecycle Management (Weight: 10%)	UI State Determinism
10	Operational Lifecycle (Weight: 10%)	UI Telemetry
11	Suite Composition	M-SUI-S-Security
12	Suite Composition	M-SUI-S-Rendering
13	Additional Evaluation Dimension	UI Architecture Dimensions
14	Additional Evaluation Dimension	The UI Doctrine
15	Additional Evaluation Dimension	Scoring Model
16	Additional Evaluation Dimension	Suite Composition

NORMATIVE PUBLICATION

Requirements, evaluation methodology, and implementation guidance

The following body preserves the substantive source standard while normalizing publication status, identity, navigation, and visual presentation.

Executive Mandate

The integration of Large Language Models with user interfaces has failed.

Current architectures treat LLMs as HTML generators, piping raw markdown or hallucinated JSX directly into the DOM via `dangerouslySetInnerHTML`. This conflation of reasoning (the model) and rendering (the UI) creates catastrophic security vulnerabilities (XSS via prompt injection), framework lock-in (React-specific components), and cognitive overload (walls of text instead of structured data).

This standard exists because:

1. The LLM is Untrusted Markup: An LLM that outputs HTML or markdown is a Cross-Site Scripting (XSS) vector. The model MUST output structured, typed *intent* (e.g., "Render an approval dialog with these options"), not *markup* (e.g., `...`).
2. Markdown is a Liability for Complex Data: Rendering network topologies, firewall blast radii, or agent state machines as markdown tables is an affront to human cognition. High-density data requires accelerated rendering (WebGPU/wgpu).
3. Framework Coupling is Architectural Suicide: Hardcoding the UI to React, Svelte, or Swift ties the system's lifespan to a framework's hype cycle. The renderer MUST be an abstract trait; the framework is a pluggable detail.
4. The Medium is Variable: An agent's output must be renderable on a browser, a Tauri desktop app, a CLI terminal, or an XR headset without rewriting the agent's output logic.

This document establishes the Mythos Semantic UI Standard (MSUIS), a comprehensive, evidence-based methodology for evaluating the protocols between AI and UI, the abstraction of renderers, and the fidelity of high-density data visualization.

Scope and Applicability

In Scope

This standard covers the evaluation of:

1. Semantic UI Protocols (LLM-to-Renderer typed payloads)
1. Typed Renderer Abstractions (multi-target: Web, Desktop, CLI, XR)
1. High-density data visualization and accelerated rendering (WebGPU, wgpu)
1. Spatial computing UI integration (WebXR, 3D engines)

1. UI state machines and deterministic rendering
1. Security boundaries between LLM output and the DOM

Out of Scope

1. General desktop architecture (covered in MYTHOS-SWE-0006)
1. AI Avatar and Persona design (covered in MYTHOS-AI-0010)
1. General accessibility (covered in MYTHOS-UX-0001)

Audience

1. Frontend engineers and UI architects
1. AI/ML engineers integrating LLMs with user interfaces
1. Security teams evaluating XSS and injection surfaces
1. Platform engineers building multi-target UI systems
1. Standards bodies seeking a reference semantic UI methodology

Definitions and Taxonomy

UI Architecture Dimensions

Dimension	Definition
Semantic Payload	A structured, typed data object (e.g., JSON/Protobuf) emitted by the LLM describing *what* should be rendered, not *how*.
Typed Renderer	An abstract interface that consumes Semantic Payloads and renders them appropriately for the target medium (Browser, Desktop, CLI, XR).
High-Density Visualization	The rendering of complex, interconnected data (e.g., network graphs, state machines) using accelerated graphics (WebGPU) rather than DOM elements.
Cyberpunk-Noir Aesthetic	A visual design philosophy prioritizing maximum data density, high contrast, minimal chrome, and accelerated rendering, optimized for operator cognition in critical infrastructure.

The UI Doctrine

> The LLM emits intent, not markup. The renderer is a trait, not a framework. Markdown is for prose; WebGPU is for infrastructure. A hallucinated tag is a security breach.

Evaluation Methodology

Scoring Model

Each capability is scored from 0 to 5.

Score	Meaning
0	Fails basic task; raw markdown/HTML, framework-coupled, DOM-heavy

Score	Meaning
1	Basic component libraries but untyped LLM output
2	Functional but lacks semantic protocols or renderer abstraction
3	Solid production performance; semantic payloads, typed renderers, basic acceleration
4	Strong performance; multi-target, WebGPU, spatial awareness
5	Best-in-class; sets the standard for deterministic, high-density semantic rendering

Weighting

Category	Weight	Rationale
Semantic Payload Contracts	25%	Unstructured LLM output is an XSS vector and a maintenance nightmare
Renderer Abstraction & Multi-Target	20%	Framework lock-in kills agility
Security & Injection Prevention	20%	The DOM is untrusted; LLM output is untrusted
High-Density & Accelerated Rendering	15%	DOM cannot render infrastructure data at scale
State & Lifecycle Management	10%	UI state must be deterministic and replayable
Operational Lifecycle	10%	UI must be governed continuously

Total: 100%

A system MUST score at least 3.0 in Semantic Payloads and Security to be considered for production use.

Evaluation Categories

Semantic Payload Contracts (Weight: 25%)

Intent vs. Markup

Does the LLM emit semantic intent (typed payloads) rather than markup (HTML/JSX)?

Score	Criteria
0	LLM outputs raw HTML/JSX strings
1	LLM outputs markdown with embedded HTML
2	LLM outputs structured JSON but with styling/layout instructions
3	LLM outputs typed semantic payloads (e.g., `{ type: "ApprovalDialog", options: [...] }`)
4	Semantic payloads defined via schema (Protobuf/JSON Schema); validated before rendering
5	Perfect separation: formally verified semantic schemas, zero markup in LLM output, compile-time payload validation

Payload Schema Governance

Are semantic payload schemas versioned and backward compatible?

Score	Criteria
0	No schema; payload structure ad-hoc
1	Implicit schema defined by renderer implementation
2	Schema defined in shared types (TypeScript)
3	Schema defined in language-agnostic format (JSON Schema/Protobuf)
4	Schema versioning with automated compatibility checks in CI
5	Perfect governance: formally verified schema evolution, zero breaking renderer changes

Renderer Abstraction and Multi-Target (Weight: 20%)

Renderer Trait

Is the UI renderer an abstract trait/interface, decoupled from the LLM and business logic?

Score	Criteria
0	Business logic directly manipulates DOM/framework components
1	Renderer abstraction exists but leaks framework types
2	Clean renderer trait; framework (React/Svelte) is a pluggable implementation
3	Multiple renderer implementations (Web, Desktop/Tauri)
4	CLI/TUI renderer for headless/terminal environments
5	Perfect abstraction: formally verified renderer trait, zero business logic awareness of rendering medium

Cross-Medium Fidelity

Can the same semantic payload render effectively across different mediums?

Score	Criteria
0	Web-only; no other targets
1	Web and mobile web (responsive)
2	Web and Desktop (Tauri/Electron)
3	Web, Desktop, and CLI/TUI
4	Web, Desktop, CLI, and native notifications
5	Perfect fidelity: Web, Desktop, CLI, and XR (WebXR/Spatial) from the same semantic payload

Security and Injection Prevention (Weight: 20%)

Markup Injection Prevention

Is the rendering pipeline immune to LLM-driven XSS/injection?

Score	Criteria
0	`dangerouslySetInnerHTML` or equivalent used for LLM output
1	Output sanitized via DOMPurify or similar
2	Allowlist of safe HTML tags enforced
3	No raw HTML rendering; semantic payloads only
4	Sandboxed rendering context (iframe/WASM) for any untrusted content
5	Perfect security: formally verified zero-markup pipeline, cryptographic attestation of payload integrity, no dynamic code execution

Payload Validation

Are semantic payloads strictly validated before rendering?

Score	Criteria
0	No validation; renderer handles malformed data
1	Basic type checking (TypeScript)
2	Runtime validation (Zod/Joi) at the boundary
3	Schema-validated payloads with descriptive errors for LLM self-correction
4	Schema validation + feedback loop to LLM for retry on schema violation
5	Perfect validation: formally verified schemas, zero invalid renders, automated LLM self-correction loop

High-Density and Accelerated Rendering (Weight: 15%)

Accelerated Graphics

Is high-density data (graphs, topologies, state machines) rendered using GPU acceleration?

Score	Criteria
0	SVG/DOM-based rendering for complex data (laggy, limited scale)
1	Canvas 2D for specific visualizations
2	WebGL for primary data views
3	WebGPU (browser) / wgpu (desktop) for compute-shader-driven layout
4	WebGPU rendering with 1M+ data points at 60fps
5	Perfect acceleration: WebGPU/wgpu, formally verified render pipelines, cyberpunk-noir density, zero frame drops under load

Data Density

Can the UI display maximum information density without cognitive overload?

Score	Criteria
0	Low density; lots of whitespace, minimal data per screen
1	Standard dashboard layouts; moderate density
2	Dense tables and charts; high information per pixel
3	Cyberpunk-Noir aesthetic: dark mode, high contrast, dense telemetry, minimal chrome

Score	Criteria
4	Adaptive density based on user role and alert state
5	Perfect density: AI-driven semantic zoom, context-aware density scaling, formally verified cognitive load limits

State and Lifecycle Management (Weight: 10%)

UI State Determinism

Is the UI state a deterministic projection of the backend state?

Score	Criteria
0	UI state managed independently in frontend; drifts from backend
1	UI state synchronized via REST polling
2	UI state derived from streaming backend events (SSE/WebSocket)
3	UI state is a pure function of backend state (local-first event sourcing)
4	State transitions are typed and deterministic; time-travel debugging supported
5	Perfect determinism: formally verified UI state machine, zero state drift, cryptographic state hashing

Operational Lifecycle (Weight: 10%)

UI Telemetry

Is the rendering pipeline observable?

Score	Criteria
0	No UI telemetry
1	Error tracking (Sentry) only
2	Performance monitoring (Core Web Vitals)
3	Semantic payload success/failure rates tracked
4	Render pipeline latency measured per payload type
5	Perfect observability: end-to-end payload-to-pixel tracing, formally verified performance bounds

The Mythos Semantic UI Suite (M-SUI-S)

Traditional UI benchmarks measure Lighthouse scores. The M-SUI-S evaluates semantic separation, injection resistance, and rendering acceleration.

Suite Composition

M-SUI-S-Security

1. XSS Injection: 100+ tests injecting malicious HTML/JS via LLM payloads to verify renderer rejection.

1. Payload Fuzzing: 50+ tests sending malformed semantic payloads to verify validation and error handling.

M-SUI-S-Rendering

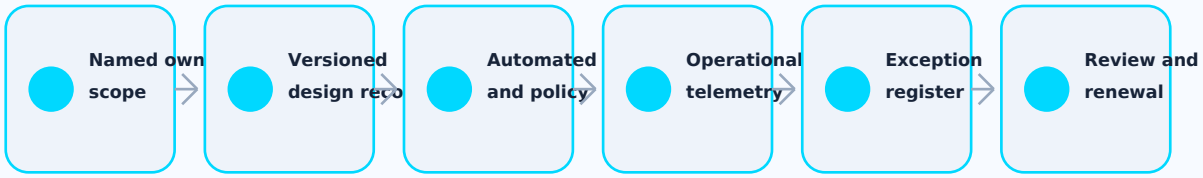
1. High-Density Load: 50+ tests rendering 1M+ data points to verify WebGPU maintains 60fps.
1. Multi-Target Fidelity: 50+ tests verifying the same payload renders correctly on Web, Desktop, and CLI.

Execution Protocol

ASSURANCE AND ADOPTION

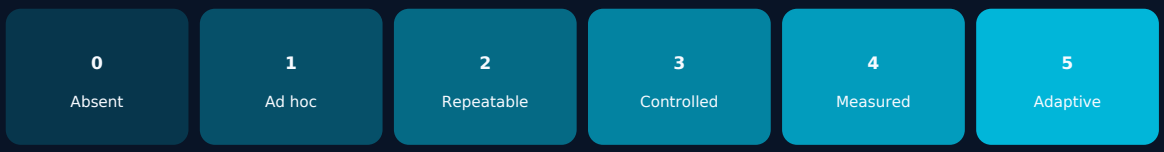
Implementation evidence and review gates

Required assurance chain



A passing score is not sufficient without current evidence. High-assurance adoption requires traceable design decisions, reproducible builds, independently reviewable tests, controlled secrets, runtime observability, failure exercises, and a dated exception register.

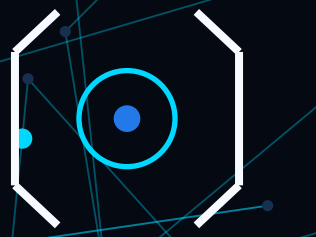
Assurance maturity spectrum



Current authority register

Authority	Version / snapshot	Applicability	Review by
Rust Project: The Rust Programming Language and Rust Reference	Rolling official documentation; book assumes Rust 1.90.0 or later and Rust 2024 Edition	High	2026-10-22
Go Project: The Go Programming Language Specification and Effective Go	Rolling official documentation snapshot	High	2026-10-22
Python Software Foundation: Python Language Reference, Packaging, Typing, and PEP Index	Rolling official documentation snapshot	High	2026-10-22
Tauri Programme: Tauri 2 Documentation	Tauri 2 rolling documentation snapshot	High	2026-10-22
Svelte Project: Svelte Documentation	Svelte 5 rolling documentation snapshot	High	2026-10-22
TLA+ Foundation: TLA+ Language and Tools Documentation	Rolling documentation snapshot	High	2027-01-24
OpenAPI Initiative: OpenAPI Specification	3.2.0	High	2027-01-24
OpenTelemetry Project: OpenTelemetry Documentation and Specification	Rolling specification snapshot; profiles public alpha and declarative configuration stable in 2026	High	2026-10-22

Research limitation. Official documentation establishes published interfaces and declared behavior. It does not prove the performance, security, interoperability, cost, or operational suitability of a specific implementation. Those claims require controlled testing and retained evidence.



ENGINEERING STANDARDS LIBRARY / SOFTWARE ENGINEERING

Software Debugging and Resilience Testing Standard

Debuggability by design, fault injection, property testing, reproducible failure analysis, and resilient recovery behavior.

PERMANENT DOCUMENT ID

MYTHOS-SWE-0009

PUBLICATION

Candidate Standard
Version 1.0.0-candidate

ASSURANCE PROFILE

Standard / Advanced
High Assurance

PERMANENT PUBLICATION IDENTITY

Software Debugging and Resilience Testing Standard

DOCUMENT ID
MYTHOS-SWE-0009

VERSION
1.0.0-candidate

STATUS
Candidate Standard

PUBLISHER
Mythos Systems

PUBLICATION DATE
2026-07-24

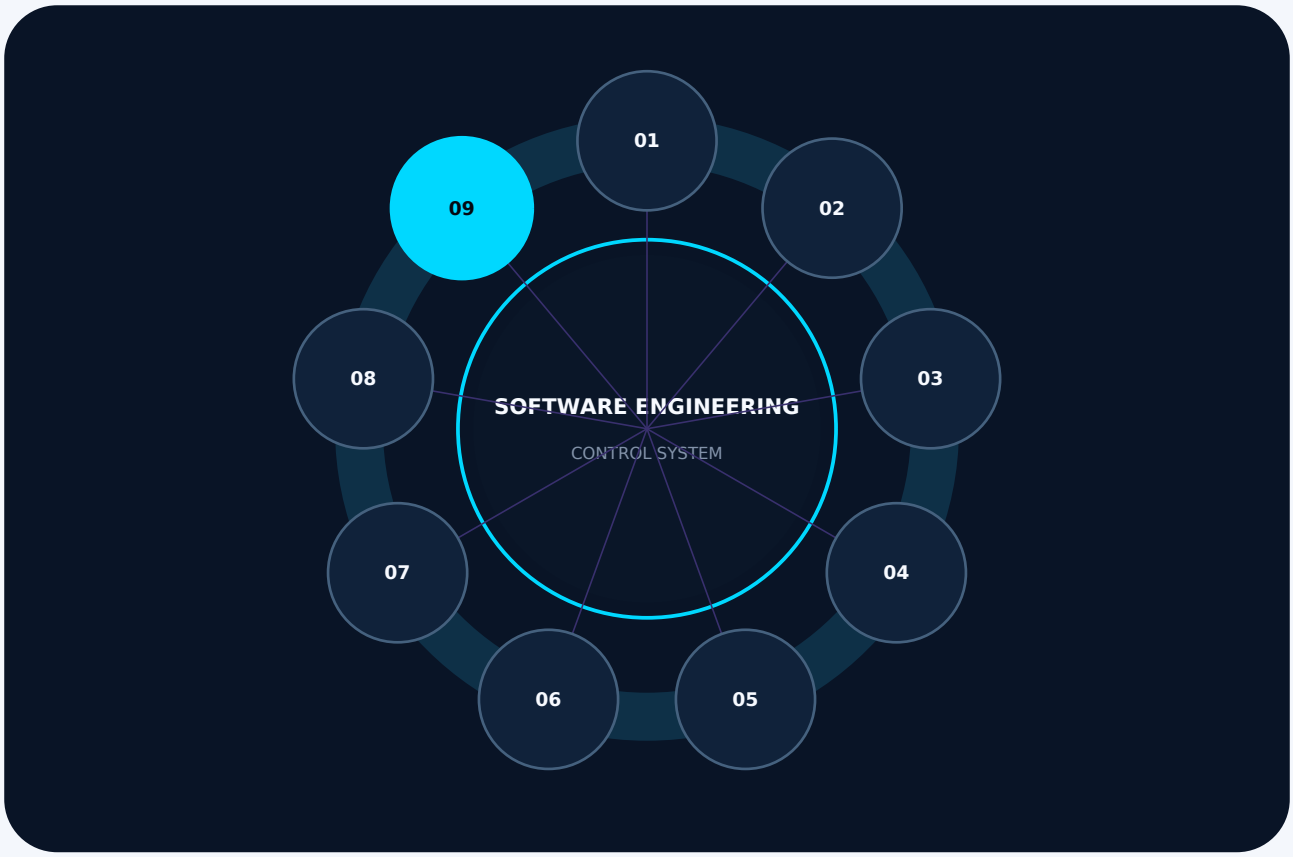
SCHEDULED REVIEW
2027-01-24

12	6	4	1
ATOMIC CRITERIA	ASSURANCE VIEWS	IMPLEMENTATION PROFILES	PERMANENT IDENTITY

Publication doctrine. This publication establishes durable guidance or requirements for its defined scope. Interpretation must preserve the permanent document identity, recorded evidence, controlled exceptions, and formal revision history.

DOMAIN CONTROL SYSTEM

MYTHOS-SWE-0009 inside the software engineering standard constellation



Connected assurance	Architecture, implementation, operations, evidence, and lifecycle are evaluated as one controlled system.
Specialization rule	Companion standards may add precision, but they cannot silently weaken this publication's requirements.

EXECUTIVE POSITION

Software Debugging and Resilience Testing Standard

The architecture of software validation has failed.

Evidence-bearing delivery path

Assurance maturity spectrum

CONTENTS

Publication navigation

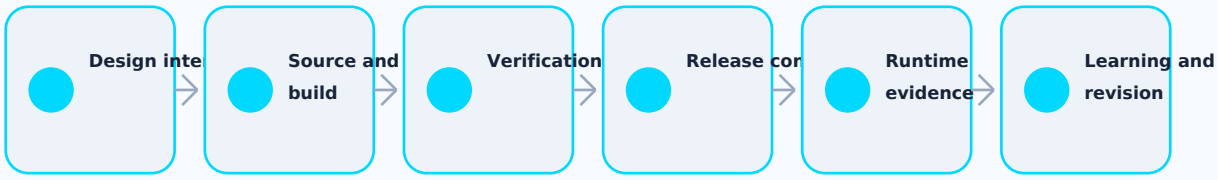
CONTROL TOPOLOGY

Assurance model and evaluation surface

The standard is assessed as an evidence-bearing system. Capability scores do not replace mandatory gates, implementation proof, or operational validation.

Capability family	Evaluation nodes
Property-Based Testing (Weight: 20%)	1
Mutation Testing Enforcement (Weight: 20%)	1
Fuzzing and Boundary Resilience (Weight: 20%)	1
Deterministic and Time-Travel Debugging (Weight: 15%)	1
Fault Injection and Chaos Engineering (Weight: 15%)	1
Test Pipeline Automation (CI/CD) (Weight: 10%)	1
Suite Composition	2
Additional Evaluation Dimension	4

Lifecycle control loop



EVALUATION INDEX

Criteria and evidence map

#	Capability family	Criterion
1	Property-Based Testing (Weight: 20%)	Invariant Proving
2	Mutation Testing Enforcement (Weight: 20%)	Test Suite Quality
3	Fuzzing and Boundary Resilience (Weight: 20%)	Input Validation
4	Deterministic and Time-Travel Debugging (Weight: 15%)	Root Cause Introspection
5	Fault Injection and Chaos Engineering (Weight: 15%)	Failure Resilience
6	Test Pipeline Automation (CI/CD) (Weight: 10%)	Enforcement
7	Suite Composition	MRTS-Mutation
8	Suite Composition	MRTS-Fault
9	Additional Evaluation Dimension	Resilience Dimensions
10	Additional Evaluation Dimension	The Testing Doctrine
11	Additional Evaluation Dimension	Scoring Model
12	Additional Evaluation Dimension	Suite Composition

NORMATIVE PUBLICATION

Requirements, evaluation methodology, and implementation guidance

The following body preserves the substantive source standard while normalizing publication status, identity, navigation, and visual presentation.

Executive Mandate

The architecture of software validation has failed.

Organizations measure the quality of their software by tracking "code coverage"—a vanity metric that proves only one thing: a function was executed. It does not prove the function computed the correct math, handled an edge case, or survived a malformed input. When a bug occurs in production, engineers attempt to debug distributed state machines by reading timestamps in centralized log aggregators, attempting to reverse-engineer race conditions from text.

This standard exists because:

1. Example-Based Testing is Insufficient: Writing `assertEqual(add(1, 2), 3)` proves the function works for 1 and 2. It tells you nothing about the other quadrillion integer combinations. Systems MUST utilize property-based testing to generate thousands of randomized inputs, proving mathematical invariants hold universally.
2. Code Coverage is a Lie: 100% line coverage does not mean 100% correctness. A test that calls a function without asserting its output counts as "covered." Test quality MUST be measured by Mutation Testing—injecting bugs into the source code and verifying the test suite catches them. If a mutant survives, the test is dead.
3. Untrusted Boundaries Must be Fuzzed: Unit tests check what the developer thought of. Fuzzers check what the attacker thinks of. Any system that ingests external data (APIs, file parsers, MCP tool outputs, LLM responses) MUST be subjected to continuous, coverage-guided fuzzing to discover memory corruption and logic bypasses.
4. Log Reading is Not Debugging: A log entry tells you *that* a microservice failed, but not *why*. Tracing a non-deterministic race condition across five services via log timestamps is impossible. Systems MUST utilize time-travel debugging and deterministic replay to inspect the exact memory state at the moment of failure.

This document establishes the Mythos Resilience & Testing Standard (MRTS), a comprehensive, evidence-based methodology for evaluating software validation pipelines against invariant proving, fault injection, and deterministic introspection.

Scope and Applicability

In Scope

This standard covers the evaluation of:

1. Property-based testing frameworks (Hypothesis, QuickCheck, proptest)
1. Mutation testing engines (Stryker, Mutmut, cargo-mutants)
1. Coverage-guided fuzzing (libFuzzer, AFL++, WebAssembly fuzzing)
1. Deterministic replay and time-travel debugging (rr, Pernosco, Replay.io)
1. Chaos engineering and fault injection testing
1. WebAssembly (WASM) sandbox resilience testing

Out of Scope

1. General Domain-Driven Design and aggregate boundaries (covered in MYTHOS-SWE-0001)
1. Formal verification of state machines (covered in MYTHOS-SWE-0007)
1. General CI/CD supply chain security (covered in MYTHOS-PLT-0001)

Audience

1. Software engineers and QA architects designing test pipelines
1. Platform engineers building fuzzing and mutation infrastructure
1. Security engineers validating input boundaries
1. SREs managing fault injection and chaos engineering
1. Standards bodies seeking a reference software validation methodology

Definitions and Taxonomy

Resilience Dimensions

Dimension	Definition
Property-Based Testing	A testing paradigm where the developer states a mathematical invariant (e.g., "decoding an encoded string yields the original string"), and the framework generates hundreds of randomized inputs to attempt to falsify it.
Mutation Testing	A method of evaluating test suite quality by automatically injecting syntactic bugs (mutants) into the source code and verifying that the unit tests fail (kill the mutant).
Coverage-Guided Fuzzing	An automated testing technique that feeds malformed, random, or unexpected data to a program, using code coverage feedback to mutate inputs that reach new execution paths.
Time-Travel Debugging	A debugging paradigm that records a program's execution (syscalls, thread schedules, memory) allowing the developer to step backward from a crash to find the root cause instruction.
Fault Injection	The deliberate introduction of errors (network latency, disk full, OOM kills) into a running system to verify its resilience and recovery mechanisms.

The Testing Doctrine

> Code coverage is a vanity metric. A passing test does not prove correctness; it only proves the code did not crash for one specific input. If a mutant survives, your test suite is an illusion. If the boundary is not fuzzed, it is compromised.

Evaluation Methodology

Scoring Model

Each capability is scored from 0 to 5.

Score	Meaning
0	Fails basic task; example-based tests only, `println!` debugging, coverage metrics
1	Basic fuzzing or property tests exist, but not enforced in CI
2	Functional test suite, but lacks mutation testing or deterministic debugging
3	Solid production performance; property testing, mutation enforcement, fuzzing, chaos engineering
4	Strong performance; time-travel debugging, WASM fault injection, formally verified invariants
5	Best-in-class; sets the standard for mathematically proven, zero-escape software resilience

Weighting

Category	Weight	Rationale
Property-Based Testing	20%	Example-based testing cannot prove mathematical invariants
Mutation Testing Enforcement	20%	Without mutation testing, high code coverage is a lie
Fuzzing & Boundary Resilience	20%	Untrusted inputs (APIs, LLMs, MCP) are the primary attack surface
Deterministic & Time-Travel Debugging	15%	Reading logs to find race conditions is architecturally impossible
Fault Injection & Chaos Engineering	15%	Systems must be tested for failure, not just success
Test Pipeline Automation (CI/CD)	10%	Resilience testing must be continuous and blocking

Total: 100%

A system MUST score at least 3.0 in Property-Based Testing and Mutation Testing to be considered for production use.

Evaluation Categories

Property-Based Testing (Weight: 20%)

Invariant Proving

Does the test suite prove mathematical invariants, or just check hardcoded examples?

Score	Criteria
0	Only example-based unit tests (`assertEqual(1, add(1, 0))`)
1	Some randomized testing, but manually generated inputs

Score	Criteria
2	Property-based framework (Hypothesis/QuickCheck) used for core logic
3	Shrinking algorithms utilized to find minimal failing cases; all domain invariants expressed as properties
4	Stateful property testing verifying multi-step state machine transitions
5	Perfect invariants: formally verified property bounds, mathematical proof of domain rules, zero edge cases unproven

Mutation Testing Enforcement (Weight: 20%)

Test Suite Quality

Is the effectiveness of the test suite measured by its ability to catch injected bugs?

Score	Criteria
0	No mutation testing; blind trust in code coverage metrics
1	Mutation testing run manually, but not enforced
2	Mutation testing run in CI, but does not block merges
3	Mutation score enforced as a CI gate; PRs blocked if mutants survive
4	Mutation testing tailored to avoid equivalent mutants; targets critical domain logic
5	Perfect enforcement: formally verified test effectiveness, zero surviving mutants in critical paths, mathematical proof of test coverage

Fuzzing and Boundary Resilience (Weight: 20%)

Input Validation

Are untrusted boundaries subjected to continuous, coverage-guided fuzzing?

Score	Criteria
0	Only developer-provided inputs tested
1	Basic dumb fuzzing (random data) with no coverage feedback
2	Coverage-guided fuzzing (libFuzzer/AFL) used on API endpoints
3	Fuzzing integrated into CI/CD; corpus continuously expanded; WASM sandboxes fuzzed
4	Fuzzing LLM tool-calling payloads and MCP server inputs for logic bypasses
5	Perfect resilience: formally verified input handling, zero memory corruption or logic bypasses, continuous distributed fuzzing

Deterministic and Time-Travel Debugging (Weight: 15%)

Root Cause Introspection

Can engineers debug race conditions and non-deterministic crashes at the instruction level?

Score	Criteria
0	Debugging via `println!` or reading log aggregators

Score	Criteria
1	Standard IDE step-debugging (forward only)
2	Core dumps analyzed for crash root causes
3	Time-travel debugging (rr/Pernosco) used to step backward from crashes
4	Distributed deterministic replay: replaying multi-service microservice interactions from recorded traces
5	Perfect introspection: formally verified state reconstruction, zero non-determinism, sub-instruction root cause isolation

Fault Injection and Chaos Engineering (Weight: 15%)

Failure Resilience

Is the system tested against infrastructure and network failures?

Score	Criteria
0	System only tested for happy-path operation
1	Manual failure testing (e.g., killing a process)
2	Automated chaos engineering on non-production environments
3	Chaos engineering in production; automated verification of system recovery (e.g., auto-restart, state reconciliation)
4	Dependency fault injection (e.g., simulating API latency, partial network partitions)
5	Perfect resilience: formally verified recovery bounds, zero state loss during fault injection, mathematical proof of self-healing

Test Pipeline Automation (CI/CD) (Weight: 10%)

Enforcement

Are resilience tests continuous and blocking?

Score	Criteria
0	Tests run manually before release
1	Tests run on PR creation, but only unit tests
2	Fuzzing and property tests run nightly, but not on PRs
3	Resilience tests (property, mutation, fuzz) run on every PR and block merges
4	Continuous fuzzing running in background 24/7; bugs auto-filed
5	Perfect automation: formally verified pipeline integrity, zero ability to bypass resilience tests, mathematical proof of release readiness

The Mythos Resilience Suite (MRTS)

Traditional benchmarks measure code coverage percentage. The MRTS evaluates invariant survival, mutation kill rate, and fault recovery.

Suite Composition

MRTS-Mutation

1. Mutant Injection: 100+ tests injecting logical bugs (e.g., changing > to >=) into PRs, verifying the CI/CD pipeline blocks the merge.
1. Property Falsification: 50+ tests running property-based testing against flawed logic, verifying the framework finds the failing input and shrinks it to the minimal case.

MRTS-Fault

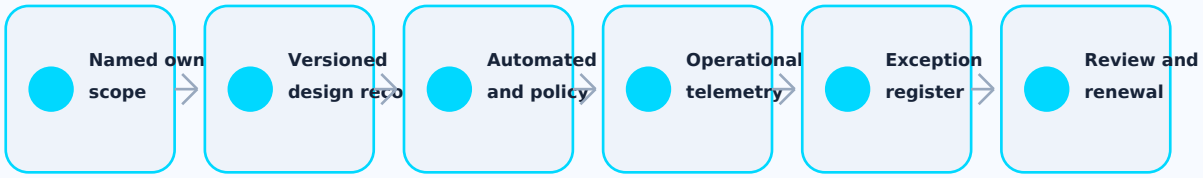
1. Fuzzer Run: 100+ tests running coverage-guided fuzzers against API endpoints and LLM tool parsers, verifying no crashes or unhandled exceptions occur.
1. Chaos Injection: 50+ tests simulating network partitions and OOM kills during complex agentic workflows, verifying the durable runtime recovers the state seamlessly.

Execution Protocol

ASSURANCE AND ADOPTION

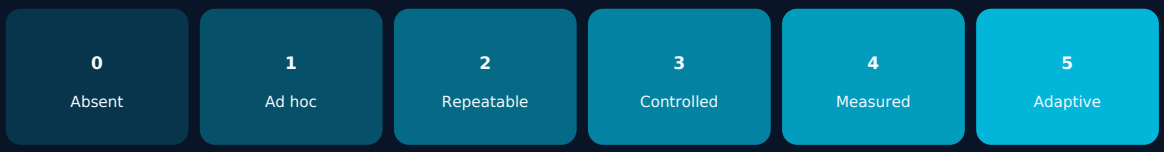
Implementation evidence and review gates

Required assurance chain



A passing score is not sufficient without current evidence. High-assurance adoption requires traceable design decisions, reproducible builds, independently reviewable tests, controlled secrets, runtime observability, failure exercises, and a dated exception register.

Assurance maturity spectrum



Current authority register

Authority	Version / snapshot	Applicability	Review by
Rust Project: The Rust Programming Language and Rust Reference	Rolling official documentation; book assumes Rust 1.90.0 or later and Rust 2024 Edition	High	2026-10-22
Go Project: The Go Programming Language Specification and Effective Go	Rolling official documentation snapshot	High	2026-10-22
Python Software Foundation: Python Language Reference, Packaging, Typing, and PEP Index	Rolling official documentation snapshot	High	2026-10-22
Tauri Programme: Tauri 2 Documentation	Tauri 2 rolling documentation snapshot	High	2026-10-22
Svelte Project: Svelte Documentation	Svelte 5 rolling documentation snapshot	High	2026-10-22
TLA+ Foundation: TLA+ Language and Tools Documentation	Rolling documentation snapshot	High	2027-01-24
OpenAPI Initiative: OpenAPI Specification	3.2.0	High	2027-01-24
OpenTelemetry Project: OpenTelemetry Documentation and Specification	Rolling specification snapshot; profiles public alpha and declarative configuration stable in 2026	High	2026-10-22

Research limitation. Official documentation establishes published interfaces and declared behavior. It does not prove the performance, security, interoperability, cost, or operational suitability of a specific implementation. Those claims require controlled testing and retained evidence.