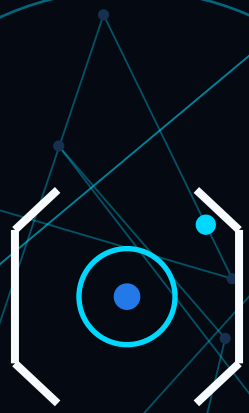


ENGINEERING STANDARDS LIBRARY / SOFTWARE ENGINEERING

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

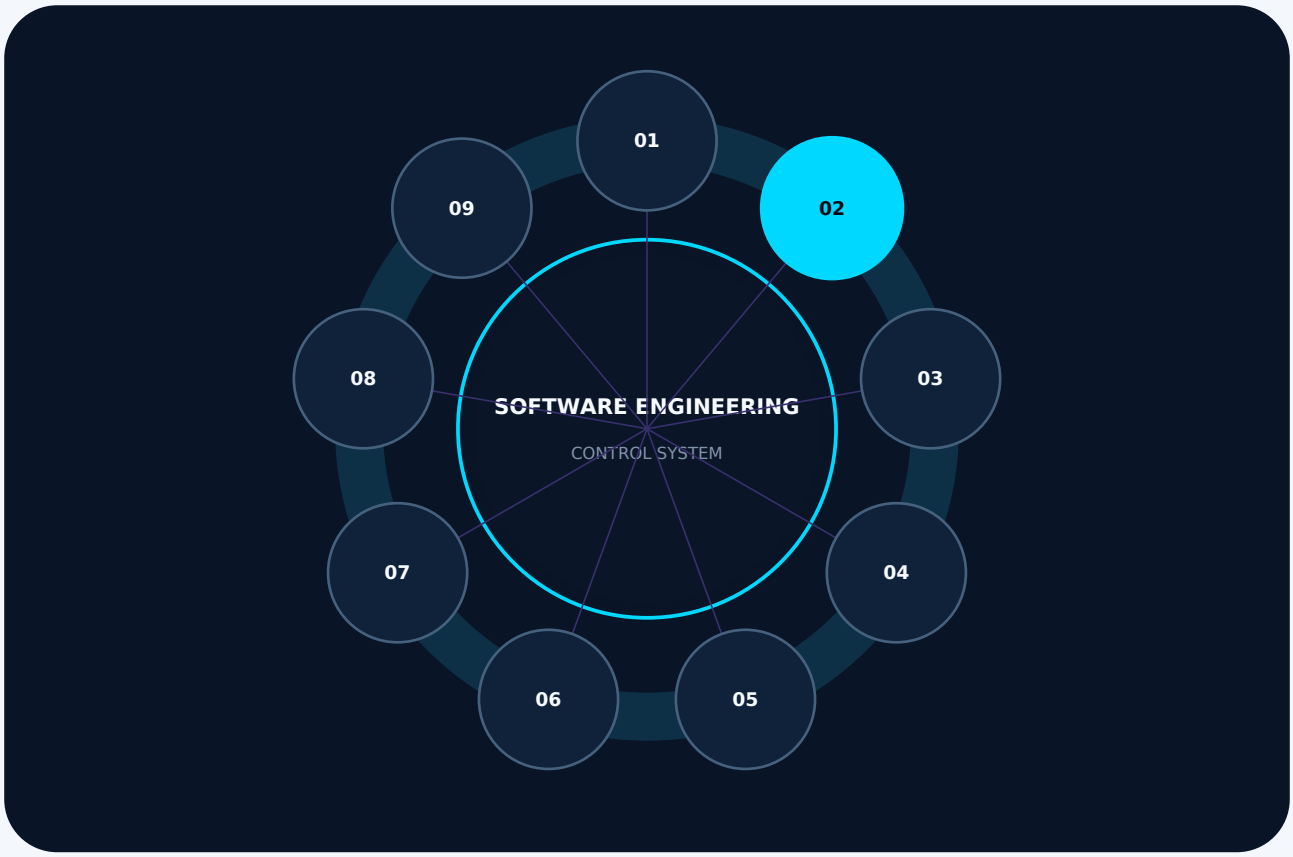
SCHEDULED REVIEW
2027-01-24

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

MYTHOS-SWE-0002 inside the software engineering standard constellation . . 3

Rust Engineering, Memory Safety, and Concurrency Standard 4

Assurance model and evaluation surface 7

Criteria and evidence map 8

Requirements, evaluation methodology, and implementation guidance 9

Executive Mandate 9

Scope and Applicability 9

In Scope 9

Out of Scope 10

Audience 10

Definitions and Taxonomy 10

Rust Dimensions 10

The Rust Doctrine 10

Evaluation Methodology 10

Scoring Model 10

Weighting 11

Evaluation Categories 11

Memory Safety and `unsafe` Governance (Weight: 25%) 11

`unsafe` Quarantine 11

Memory Allocation Discipline 11

Concurrency and Async Discipline (Weight: 20%) 12

Bounded Concurrency 12

Cancellation and Shutdown 12

Error Handling and Panics (Weight: 20%) 12

Panic-Free Execution Paths 12

Typed Error Architecture 13

Type System and Invariant Enforcement (Weight: 15%) 13

Type-State Modeling 13

Crate Architecture and Dependencies (Weight: 10%) 13

Dependency Minimalism 13

Tooling and Linting (Weight: 10%) . . . 14
Clippy and Formatting . . . 14

The Mythos Rust Engineering Suite (MRES) . . . 14
Suite Composition . . . 14
MRES-Safety . . . 14
MRES-Concurrency . . . 14
Execution Protocol . . . 14

Implementation evidence and review gates . . . 15
Current authority register . . . 16

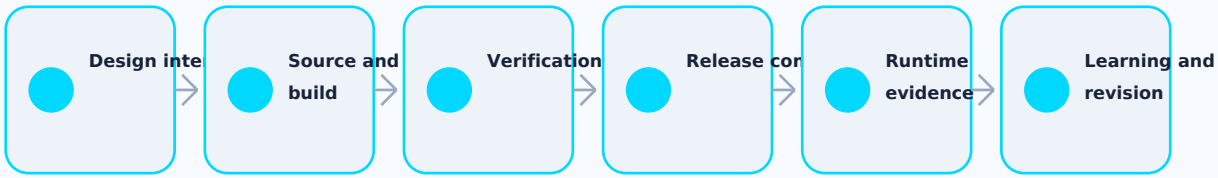
CONTROL TOPOLOGY

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

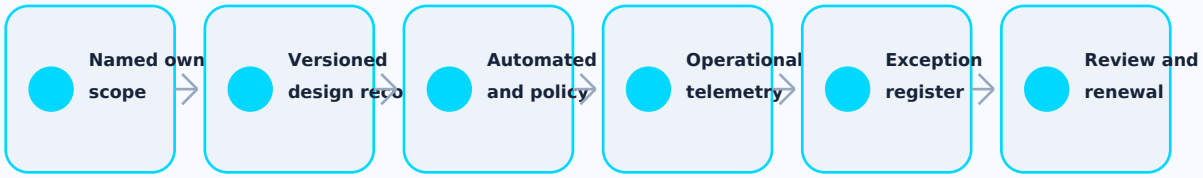
1. Unbounded Queue Detection: 50+ tests scanning for unbounded channels.
1. Shutdown Race: 50+ tests abruptly cancelling tasks to verify no resource leaks or deadlocks.

Execution Protocol

ASSURANCE AND ADOPTION

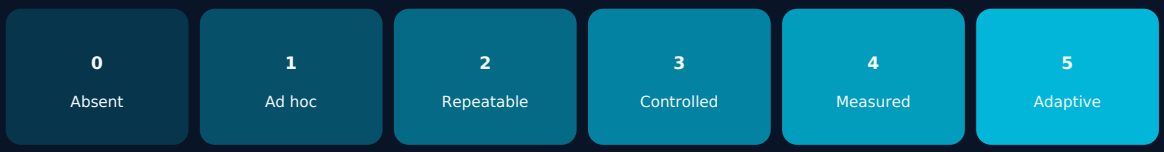
Implementation evidence and review gates

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.