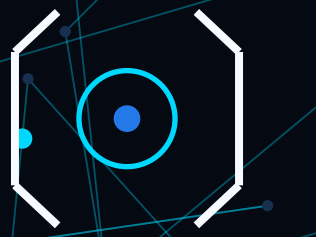


ENGINEERING STANDARDS LIBRARY / SOFTWARE ENGINEERING

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

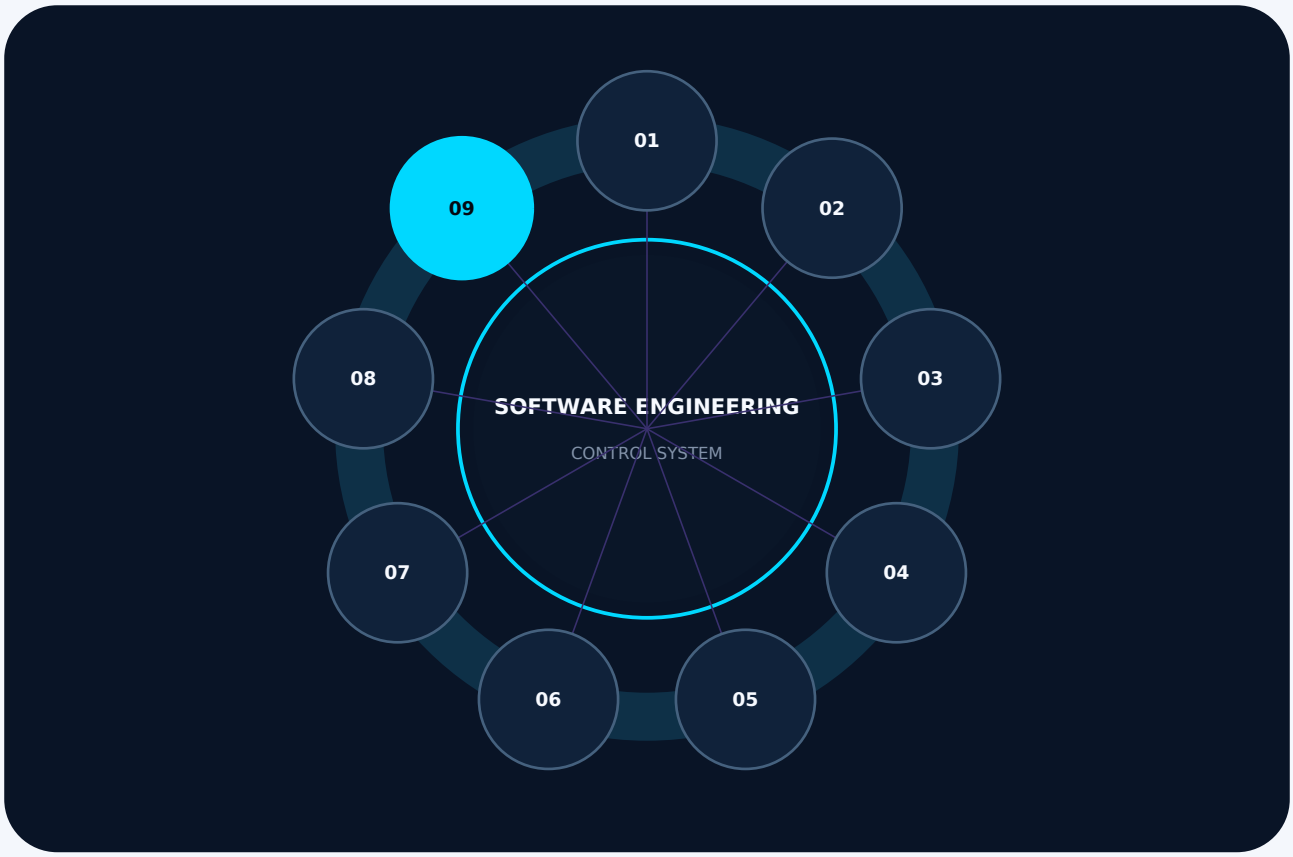
SCHEDULED REVIEW
2027-01-24

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

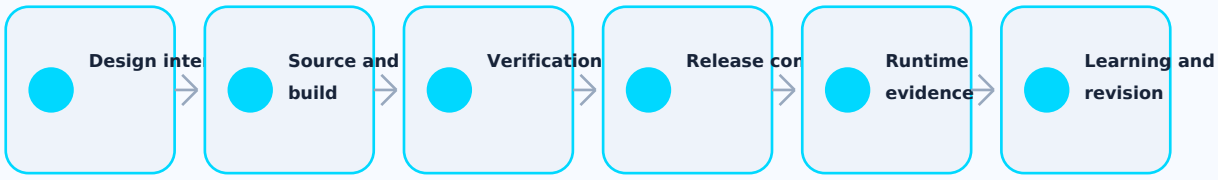
CONTROL TOPOLOGY

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

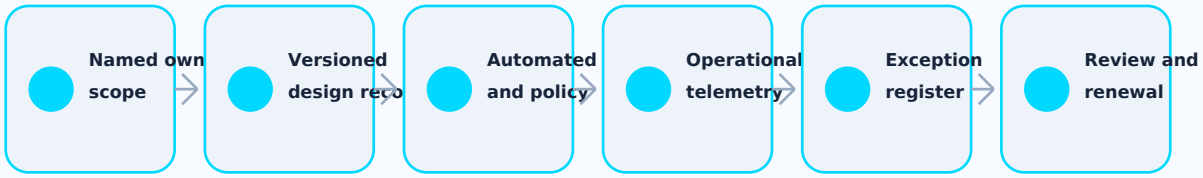
1. Fuzzer Run: 100+ tests running coverage-guided fuzzers against API endpoints and LLM tool parsers, verifying no crashes or unhandled exceptions occur.
1. Chaos Injection: 50+ tests simulating network partitions and OOM kills during complex agentic workflows, verifying the durable runtime recovers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

Execution Protocol

ASSURANCE AND ADOPTION

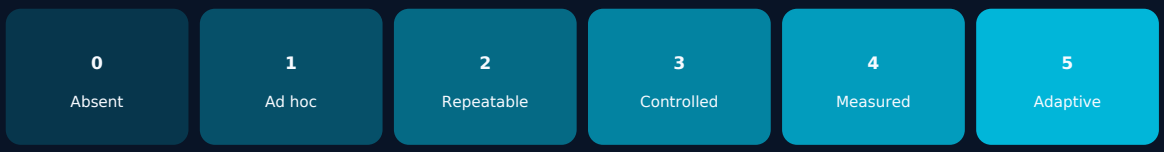
Implementation evidence and review gates

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.