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

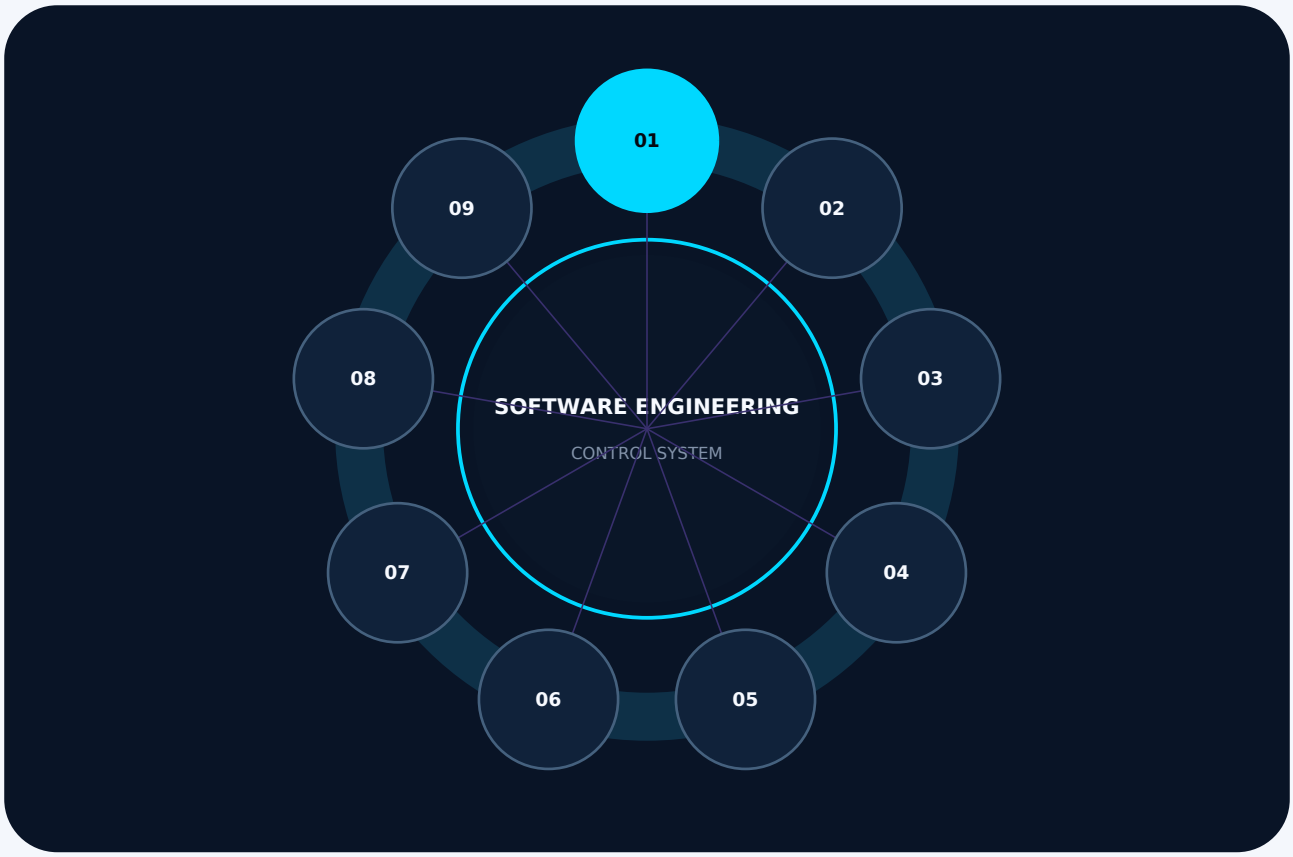
SCHEDULED REVIEW
2027-01-24

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

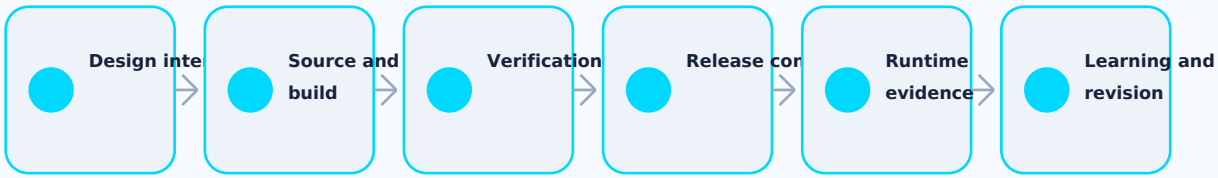
CONTROL TOPOLOGY

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

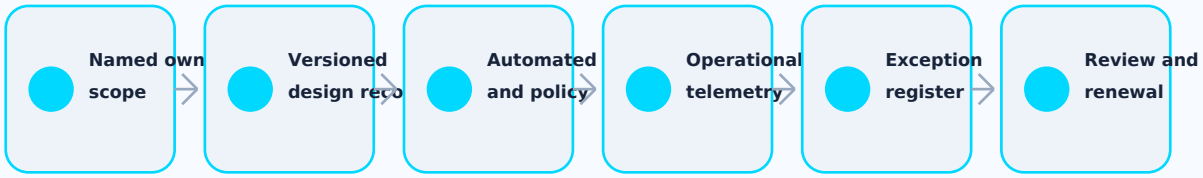
1. Invariant Violation: 100+ tests attempting to put aggregates into invalid states.
1. Contract Compatibility: 50+ tests verifying backward compatibility of API contracts across versions.

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.