MODIVX STANDARD DEFINITION
INTEROP STANDARD

Protocol version	1.0
Document version:	2.0
Status:	Release Candidate
Category:	Standards Track
Date:	2026-01
Author:	

The Seven-Layer MODIVX Specification Stack

1. MLD Mathematical Core (MC)
2. Binding Specification
3. Wire Protocol
4. Wire Output Envelope
5. JSON Schema
6. Implementation Profile
7. **Interop Standard**

Status of This Document (Tests and Certification)

The Interop Standard defines the conformity and interoperability tests that must be passed to demonstrate correct implementation of the specification. Rationale for the final layer position: Tests can only verify what has been previously defined. They do not replace a definition; rather, they merely provide evidence that existing standards are being met.

This document constitutes the MODIVX **DCO Profile** only.

A) STD-0 – Entry Point

A.1 Purpose

The MLD Standard defines an implementable and interoperable standard for model-structure diagnostics based on:

- metalogical equivalence-class formation,
- dimension-wise ordinal diagnosis,
- finite/budgeted approximations (Horizon),
- deterministic execution including a standardized error/disclosure mechanism,
- well-founded adaptation dynamics (Pareto progress),
- (optional) meta-adaptive schema control.

The objective is vendor- and language-independent implementability of:

1. MLD Engine (compute core),
2. MLD Analyzer (orchestration + findings),
3. MLD Wire protocol (interoperability).

A.2 Standard Scope

This standard normatively specifies:

1. Semantics & correctness framework (MC)
2. Effective execution and interface specifications (ImplSpecs)
3. Engine Binding (API/ABI + disclosures + testability)
4. Implementation Profile (control flow, determinism, logging/mapping)
5. Wire Protocol (messages + state machine)
6. Conformance (levels, requirements, tests)

A.3 Standard Layer Model

A.3.1 MLD Engine

Implements the MC/ImplSpecs operators:

- Enc/Canon/Key (EqClass)
- Img / Reach / States
- δ_i (diagnostic operator per dimension)
- behavior predicates Stable/Reactive/Sensitive (Horizon-relative)
- A_i (adaptation per dimension)
- RunAdaptation (trajectory)
- Observe / NextTheta / RankTheta (meta)

The engine is a deterministic function of the inputs including BindParams.

A.3.2 MLD Analyzer

Orchestrates engine calls according to the Implementation Profile.

The analyzer produces Findings/Errors/Wire outputs exclusively according to normative rules (canonical ordering, complete disclosures, no hidden approximations).

A.3.3 MLD Wire

Normatively specifies message exchange between Client/Analyzer (and optionally the engine).

A.4 Normative References

The following documents are normative, in priority order:

1. MLD Mathematical Core (MC)
2. MOVIDX Implementation Specs
3. MOVIDX Binding Specification
4. MOVIDX Implementation Profile
5. MOVIDX Wire Protocol

In case of conflict:

- The MC prevails over all documents.
- Implementation Specs operationalize the MC (effective approximations).
- Binding/Profile/Wire operationalize interoperability + disclosure.

A.5 Compliance Levels

Conformance is defined in levels. A system may be partially conformant.

L1 – Wire Interoperability

- wire messages are syntactically valid
- deterministic message fields (canonical ordering)
- standardized error mapping

L2 – Deterministic Findings

- engine/analyzer deterministic
- canonical ordering of all iterated structures

- complete disclosures: BindParams, Horizon/Approx-Policy, registry_snapshot_id, fingerprints

L3 – Adaptive Compliance

- progress/Pareto admissibility
- Neutral Selection Rule (3-stage)
- termination conditions (fixpoint, cycle, max steps)
- trajectory + step logging

L4 – Meta Compliance

- meta-observation from trajectory/logs
- schema switching via NextTheta
- rank monotonicity / locking mechanism
- meta-termination guaranteed

A.6 Global Determinism Requirement

All standardized outputs are deterministic:

Same inputs (including BindParams) ⇒ byte-identical normalized output.

Non-deterministic operations (hash iteration, unsorted sets/maps) are prohibited unless canonical ordering is enforced.

A.7 Disclosure / Approximation

Every approximation must be disclosed as input and/or output:

- Horizon: Nmax, CandidateBudget, TimeBudget
- approx_policy / run_mode / tie_break_policy
- registry_snapshot_id
- config_fingerprint
- context_fingerprint
- witness_set_id (if max_collision_rate > 0 is disclosed)
- bind_params_fingerprint
- for L3/L4 additionally:
    - MaxSteps
    - cycle definition
    - ThetaImpl manifest + RankTheta

A.8 Conformance

A system is conformant at level Lk if and only if it passes the Conformance Suite tests for Lk.

Proof of conformance:

Evidence of passed conformance tests MUST be stored as a machine-readable conformance report.

The conformance report MUST at minimum contain:

- implementation identifier (name/vendor) and version/build ID,
- targeted compliance level Lk,
- applicable standard/schema version(s),
- test list with PASS/FAIL per test ID,
- overall result (PASS only if all tests pass).

Pass criterion:

A system is conformant at level Lk if and only if it passes all tests defined as mandatory for Lk in the test catalog; “partial” results do not constitute conformance.

B) Requirements Register (L1–L4)

Notation:

- MUST = mandatory for conformance (of the respective level)
- SHOULD = recommended
- Test ID references the Conformance Suite

Test coverage:

Every requirement of type MUST (per level) MUST be mapped to at least one test ID in the Conformance Suite.

B.1 (L2) – Determinism, Ordering, Errors, Coverage, Disclosure

REQ-0001 Deterministic Output

Level: L2 — Type: MUST

For identical inputs (EqClass, context Z, BindParams, registry snapshot), the normalized output MUST be byte-identical.

Test: T-DET-001

REQ-0002 Canonical Ordering: Dimensions

Level: L2 — Type: MUST

Dimensions MUST be iterated and emitted in the manifested DimOrder.

Test: T-ORD-001

REQ-0003 Canonical Ordering: EqClass Sets

Level: L2 — Type: MUST

All sets of EqClass (e.g., candidates, reach sets) MUST be deterministically sorted by CanonKey.

Test: T-ORD-002

REQ-0004 Canonical Ordering: Maps

Level: L2 — Type: MUST

Maps/dictionaries in outputs MUST be ordered lexicographically by key or serialized as a deterministic sequence.

Test: T-ORD-003

REQ-0005 Canon Contract: Determinism

Level: L2 — Type: MUST

Canon(Enc(phi)) and/or key_eqclass(phi) MUST be deterministic.

Test: T-CAN-001

REQ-0005A Canon Contract: Collision Monitor Disclosure

Level: L2 — Type: MUST

If max_collision_rate > 0 is used/disclosed, witness_set_id MUST be disclosed.

Test: T-CAN-002

REQ-0006 Error Model: Result[T]

Level: L2 — Type: MUST

Partial functions MUST be modeled as total functions using Result[T] (Ok | Err with standardized ErrorCodes).

Test: T-ERR-001

REQ-0007 Separation: State vs Error/Confidence Metadata

Level: L2 — Type: MUST

State MUST be strictly separated from Errors/Confidence/Evidence; errors MUST NOT be encoded via state degradation.

Test: T-DIAG-001

REQ-0008 Coverage: Candidate image non-empty

Level: L2 — Type: MUST

For each dimension i, Img_i_impl MUST be non-empty.

Test: T-COV-001

REQ-0009 Coverage: Self-inclusion

Level: L2 — Type: MUST

For each dimension i: [phi] $\in$ Img_i_impl([phi], z).

Test: T-COV-002

REQ-0010 No-Op Perturbation

Level: L2 — Type: MUST

For each dimension i, there exists a canonical no-op perturbation p0,i such that $Apply_i$(p0,i,[phi]) = Ok([phi]).

Test: T-COV-003

REQ-0011 Disclosure: BindParams present

Level: L2 — Type: MUST

Outputs MUST include BindParams/Horizon/approx_policy and registry_snapshot_id; config_fingerprint is optional.

Test: T-DISC-001

B.2 (L1) – Wire Interoperability

REQ-0101 Wire Message Validity and Alias Equivalence of Message Types

Level: L1 — Type: MUST

All wire messages MUST conform to the MLD-Wire schema, including all required fields and all if/then consistency constraints (e.g., XOR rules, terminal ERROR rules, required disclosures).

Where aliases are used (e.g., DIAGNOSE_REQUEST ≙ DIAG_REQUEST, DIAGNOSTIC_FINDING ≙ DIAG_RESULT, META_DIAGNOSTIC_FINDING ≙ META_RESULT), this is representation-only; semantics and validation rules remain identical.

Test: T-WIRE-001

REQ-0102 Wire Deterministic Fields

Level: L1 — Type: MUST

Wire sequences/fields (findings, dimension blocks, candidate lists) MUST be canonically ordered.

Test: T-WIRE-002

REQ-0103 Wire Error Mapping

Level: L1 — Type: MUST

Engine/analyzer errors MUST be mapped to standardized wire error codes.

Test: T-WIRE-003

B.3 (L3) – Adaptive Compliance (Progress, Selection, Trajectory)

REQ-0201 Progress Criterion (Pareto admissibility)

Level: L3 — Type: MUST

An update $[\varphi] \to [\psi]$ is admissible only if:

1. no degradation in any dimension (component-wise $\leq_i$), and

2. at least one strict improvement in at least one dimension.
   Test: T-ADAPT-001

REQ-0202 No Total Order on Profiles

Level: L3 — Type: MUST

No total order is introduced on state profiles. Candidates are selected deterministically only via Pareto maximality + Neutral Selection Rule.

Test: T-ADAPT-002

REQ-0203 Neutral Selection Rule (3-stage)

Level: L3 — Type: MUST

If multiple admissible candidates exist, Select MUST be implemented as:

1. Pareto-maximal (admissible set),
2. minimal EditCost,
3. final tie-break: minimal CanonKey.
   Test: T-ADAPT-003

REQ-0204 CanonKey only final tie-break

Level: L3 — Type: MUST

CanonKey MUST be used exclusively as the final tie-breaker and not as a ranking substitute.

Test: T-ADAPT-004

REQ-0205 Adaptation step totality

Level: L3 — Type: MUST

Adapt([φ], z) is total: if $\mathcal{U}$([φ], z)=∅, then Adapt([φ], z)=[φ].

Test: T-ADAPT-005

REQ-0206 Trajectory termination controls

Level: L3 — Type: MUST

The effective trajectory MUST implement termination conditions: fixpoint, cycle detection, MaxSteps.

Test: T-ADAPT-006

REQ-0207 Step logging (object level)

Level: L3 — Type: MUST

Each step MUST log at least: φ_n, selected dimension i_n, φ_{n+1}, Horizon H, progress justification / admissible count.

Test: T-ADAPT-007

REQ-0208 Selection semantics: no profile ranking

Level: L3+ — Type: MUST

Tie-breakers/selection order serve exclusively deterministic choice within an admissible set (e.g., Pareto maximal) and MUST NOT be interpreted as a semantic total order/aggregation of profiles.

Test: T-SEL-001

B.4 (L4) – Meta Compliance (Schema Switching)

REQ-0301 Finite Theta set

Level: L4 — Type: MUST

A finite set of implemented schemata Θ_impl exists, and a start schema Θ0.

Test: T-META-001

REQ-0302 Observe determinism

Level: L4 — Type: MUST

Meta-observation Observe(trajectory, logs) is deterministic.

Test: T-META-002

REQ-0303 NextTheta determinism

Level: L4 — Type: MUST

NextTheta(Θ, 𝔖) is deterministic (Result scheme).

Test: T-META-003

REQ-0304 Rank monotonicity / non-increasing progress constraint

Level: L4 — Type: MUST

Schema switching is admissible only if RankTheta does not increase (per specification).

Test: T-META-004

REQ-0305 Meta termination / lock

Level: L4 — Type: MUST

After reaching the last/minimal rank level, a locking mechanism applies (no further switching).

Test: T-META-005

REQ-0306 Meta logging

Level: L4 — Type: MUST

Meta execution logs: observations, schema switching, RankTheta values.

Test: T-META-006

REQ-0307 RankTheta direction disclosure

Level: L4 — Type: MUST

If rank_in/rank_out are used, the RankTheta monotonicity direction MUST be disclosed as rank_theta_direction.

Test: T-RANK-001

C) Conformance Test Catalog (L1–L4)

Normative (in combination with the Requirements Register)

Test artifacts:

- For each test listed here, a corresponding test artifact (input + expected output) MUST exist.
- The expected output is the normative reference (“Golden Output”) and MUST be checked against the normalized output.

L1 Tests

T-WIRE-001 Wire validity

REQs: REQ-0101

Check: output satisfies the wire schema (envelope)

T-ALIAS-001 Alias equivalence

REQs: REQ-0101

Check: if messages are serialized using alias type names, schema/rule validation remains equivalent (same required fields, same XOR/terminal rules).

T-WIRE-002 Wire ordering

REQs: REQ-0102

Check: wire sequences are canonically ordered

T-WIRE-003 Wire error mapping

REQs: REQ-0103

Check: error codes are mapped correctly

L2 Tests

T-DET-001 Determinism

REQs: REQ-0001

Check: identical input ⇒ byte-identical normalized output

T-ORD-001 DimOrder

REQs: REQ-0002

Check: dimensions appear in DimOrder

T-ORD-002 EqClass ordering

REQs: REQ-0003

Check: candidate/reach lists are CanonKey-sorted

T-ORD-003 Map ordering

REQs: REQ-0004

Check: maps are deterministically serialized

T-CAN-001 Canon determinism

REQs: REQ-0005

Check: key_eqclass/CANON is stable

Additional check: canon_contract.canon_format is one of the normative values {FULL, HASHED}.

T-CAN-002 Collision monitoring disclosure

REQs: REQ-0005A

Check: if max_collision_rate > 0 is disclosed in output, witness_set_id is present.

T-ERR-001 Result scheme

REQs: REQ-0006

Check: partial functions return Result[T] (Err instead of hidden failure)

T-ERR-002 Terminal ERROR encoding

REQs: REQ-0101

Check: for messages type="ERROR": len(errors)=1 and payload is not present.

Check: for result messages (DIAG_RESULT, ADAPT_RESULT, META_RESULT): len(errors)=0.

T-DIAG-001 Separation

REQs: REQ-0007

Check: state is ordinal; errors/confidence separate (no state downgrade)

T-COV-001 Non-empty Img

REQs: REQ-0008

Check: Img_i_impl is non-empty

T-COV-002 Self-inclusion

REQs: REQ-0009

Check: $\varphi \in Img_i$_impl

T-COV-003 No-op perturbation

REQs: REQ-0010

Check: NoOp exists and Apply(NoOp)=Ok(phi)

T-DISC-001 Disclosure

REQs: REQ-0011

Check: bind_params/horizon present and registry_snapshot_id present; config_fingerprint optional; context_fingerprint present; bind_params_fingerprint present.

T-WIRE-000 Schema validation

REQs: REQ-0101

Check: every wire message is validated against the normative MLD ENVELOPE JSON SCHEMA v1.0 (including required and if/then conditions).

Result: FAIL if any message is schema-invalid.

L3 Tests (Adaptation)

T-ADAPT-001 Pareto admissibility

REQs: REQ-0201

Check: each applied update step satisfies non-degradation + at least one strict improvement

T-ADAPT-002 No profile total order

REQs: REQ-0202

Check: output contains no aggregated profile number / no total-order encoding; selection uses Pareto + tie-break

T-ADAPT-003 Neutral selection rule

REQs: REQ-0203

Check: selection protocol shows correct 3-stage selection (Pareto-max → EditCost → CanonKey)

T-ADAPT-004 CanonKey only final tie-break

REQs: REQ-0204

Check: CanonKey is not used before EditCost

T-ADAPT-005 Adapt step totality

REQs: REQ-0205

Check: if U empty ⇒ result=phi

T-ADAPT-006 Trajectory termination

REQs: REQ-0206

Check: fixpoint/cycle/MaxSteps are detected correctly

T-ADAPT-007 Step logging

REQs: REQ-0207

Check: logs contain at least phi_n, i_n, phi_n1, H, admissible_count / progress justification

T-SEL-001 Selection is not ranking

REQs: REQ-0208

Check: documentation/disclosures declare tie-breaker/selection order as deterministic selection only; no semantic profile ranking is asserted or encoded.

L4 Tests (Meta)

T-META-001 Finite Theta set

REQs: REQ-0301

Check: ThetaImpl manifest + Theta0 present

T-META-002 Observe determinism

REQs: REQ-0302

Check: Observe(trajectory, logs) deterministic

T-META-003 NextTheta determinism

REQs: REQ-0303

Check: NextTheta deterministic (Result scheme)

T-META-004 Rank monotonicity

REQs: REQ-0304

Check: schema switching only if RankTheta does not increase

T-META-005 Meta termination lock

REQs: REQ-0305

Check: lock engages after the last level (no further switching)

T-META-006 Meta logging

REQs: REQ-0306

Check: meta logs contain observation, theta switching, ranks

T-RANK-001 Rank monotonicity

REQs: REQ-0307

Check: rank_theta_direction present.

Check: if NON_INCREASING: rank_out <= rank_in. If NON_DECREASING: rank_out >= rank_in.

D) Output Normalizer Spec

D.1 Purpose and Scope

This specification defines an output normalizer that establishes a canonical normal form for MODIVX/MLD outputs. The objective is determinism and interoperability of the conformance proof.

The output normalizer is used exclusively for conformance verification (conformance-by-testing), in particular for producing and checking “Expected Outputs” (Golden Outputs) and for comparing implementation outputs.

Normative objective:

Two outputs are considered equivalent if their normalized form is byte-identical.

D.2 Clarification of the Comparison Basis

D.2.1 Comparison Object

Normalization applies to the output scope defined as the subject of verification by the Conformance Test Catalog. The standard MUST clearly specify whether comparison is applied to:

- the entire wire envelope, or

- payload substructures, or
- defined substructures.

D.2.2 Validity Preconditions

An output that does not satisfy the formal wire/schema requirements is invalid. In this case, the test case MUST be evaluated as FAIL; “best effort” normalization of invalid outputs is prohibited.

D.3 Canonical Serialization

D.3.1 Encoding

The normal form MUST be encoded as UTF-8.

D.3.2 Whitespace

The normal form MUST be serialized without non-semantic whitespace variations. Pretty-printing/formatting whitespace is not permitted.

D.3.3 Escape Rules

String escapes and control characters MUST be encoded in a canonical and uniquely reproducible form.

D.4 Deterministic Ordering (Ordering Semantics)

D.4.1 Object/Map Key Ordering

For all JSON objects or map structures: keys MUST be ordered deterministically using a normative comparison relation. The comparison relation MUST be unambiguous (e.g., lexicographic order based on a defined Unicode/ASCII rule).

D.4.2 Arrays / Lists

The standard MUST distinguish two categories:

1. Sequences (order-significant)
   Arrays whose order is semantically relevant MUST NOT be reordered by the normalizer.
2. Sets (order-insignificant)
   Arrays interpreted as sets MUST be sorted canonically by the normalizer. A canonical sort key MUST be defined (e.g., CanonKey, ID, or lexicographic serialization).

Note:

If a list is required to be deterministic/canonical by the standard (e.g., candidate lists, EqClass lists, reach lists), its ordering rule MUST be explicitly specified; the term “deterministic” alone is insufficient.

D.5 Numeric Normalization

Numeric values MUST be serialized in a uniquely canonical form, including at least:

- normalization of zero: -0 is prohibited (or must be converted to 0),
- exponent notation: if permitted, it MUST be normalized; otherwise it MUST be disallowed,
- precision/rounding: rounding/precision rules MUST NOT be implementation-dependent; if relevant numeric quantities occur, a precise normalization MUST be specified,
- special values: NaN/Infinity must either be explicitly prohibited or normatively defined (recommended: prohibition).

D.6 String Normalization (Unicode)

String values MUST be in a normative Unicode normal form (e.g., NFC). The objective is that semantically identical strings do not diverge due to differing Unicode representations.

Control characters and escape sequences MUST be treated canonically.

D.7 Null, Missing Fields, and Optional Attributes

The standard MUST explicitly specify whether null and “missing field” differ semantically. For conformance checks:

- missing fields and null are not automatically equivalent unless normatively specified,
- the normalizer MUST NOT erase semantic differences.

D.8 Operationalization (Normative Algorithm)

The normalizer MUST be specified as a deterministic transformation function. A valid minimal algorithm is:

1. Validate input: output must be schema/wire-valid (see D.2.2).
   Recursively normalize:
   → objects: sort keys, normalize values
   → sequence lists: preserve order, normalize elements
   → set lists: canonically sort, normalize elements

   - → numbers: normalize per D.5
   - → strings: normalize per D.6
2. Canonically serialize (D.3) into a byte sequence.
3. Compare by byte equality.

D.9 Verifiability and Reference Artifacts

To eliminate interpretive ambiguity, at least one of the following MUST be provided:

- a reference implementation of the normalizer, or
- normative normalizer test vectors (normalizer input → expected normal form), or
- normative hash values of the normalized reference outputs.

D.10 Conformance Criterion (Link to Test Catalog)

A conformance test whose expected outcome is PASS is passed only if the normalized output byte sequence is byte-identical to the associated Golden Output.

D.11 Canonical Sort Key Definitions

D.11.1 Principle

For all structures required to be deterministic by the standard:

- if a list or map is semantically a set, it MUST be converted to a canonical order,
- if a list is semantically a sequence, the normalizer MUST NOT change its order,
- every canonical sort MUST be defined by a normative sort key.

D.11.2 Canonical Sort Key

Unless specified otherwise, the sort key is lexicographic order of a canonical key representation:

1. primary: canon_key (if present)
2. secondary: deterministic normalized serialization of the element (D.3–D.8)
3. tertiary: explicitly defined ID fields (if provided by the standard)

If keys compare equal, a deterministic tie-breaker MUST be used (secondary/tertiary keys) so that a total order results.

D.11.3 Canonical Ordering for Standard-Defined Lists (Mandatory)

The following list structures MUST be canonically ordered:

- EqClass/equivalence-class related lists: candidates, reach lists, canon/equivalence-class lists MUST be sorted by canon_key (or an equivalent derived canonical key per MC/Specs).
- candidate/reach lists MUST be canonically sorted to eliminate implementation-dependent iteration order (set/map iteration).
- map serialization: JSON objects/maps MUST be key-sorted lexicographically per D.4.1.

D.11.4 Canonical Ordering for Lists with Normative Order (DimOrder)

For lists whose order is explicitly normative (e.g., DimOrder):

- order MUST be preserved exactly as specified by the normative DimOrder definition,
- the normalizer MUST treat the list as a sequence and MUST NOT reorder it,
- order is part of the canonical form for comparisons.

D.11.5 Explicit Non-Sorting (Order-significant sequences)

The following sequence types MUST NOT be canonically sorted:

- trajectory/step lists if their order is temporal/process semantic,
- logs if their order is part of the state/control logic,
- any list described by the standard as an “ordered sequence” or equivalent.

D.11.6 Normative Note on Testability

If a conformance test requires sorting/ordering (e.g., “CanonKey-sorted”, “DimOrder”), the test artifact expected/output.json MUST already contain that order in normalized form.

D.11.7 Sorting Matrix

Where the standard is not unambiguous about whether a list is a set or a sequence, the standard MUST explicitly specify it (otherwise interop is not formally provable).

Output structure type	Semantic category	Normalizer action	Sort key / rule
JSON Object / Map	Set (order-insignificant)	MUST sort keys	Lexicographic key ordering per D.4.1
EqClass element list	Set	MUST sort elements	Primary: canon_key; Secondary: canonical serialization (D.3–D.8)
Candidate list	Set	MUST sort elements	Primary: canon_key; Secondary: canonical serialization
Reach list	Set	MUST sort elements	Primary: canon_key; Secondary: canonical serialization
Canon/Key EqClass list	Set	MUST sort elements	canon_key (or normatively derived canonical key)
Maps deterministically serialized (Test T-ORD-003)	Set	MUST sort keys	Same as JSON Object / Map

DimOrder list (dimensions)	Sequence (order-significant)	MUST preserve order	Exact order per normative DimOrder
Wire sequences required to be canonically ordered (Test T-WIRE-002)	Sequence (normatively ordered)	MUST preserve order	Order per Wire rules / Envelope specification
Trajectory / Step list (adaptation)	Sequence	MUST preserve order	Process/temporal order is semantic
Logs (object-level logging)	Sequence	MUST preserve order	Order is part of the proof/evidence
Error list / error stack (if present)	Sequence or Set (must be normatively fixed)	MUST preserve order OR MUST sort	If set: sort by (error_code, location/path/field, canonical serialization); otherwise preserve
Disclosure container	Set (fields)	MUST sort keys	JSON key ordering; no semantic reordering
Fingerprint / snapshot fields	Atomic	MUST NOT modify	No normalization beyond canonical string representation
Numeric fields	Atomic	MUST canonicalize	Per D.5 (no -0, no NaN/Inf, defined notation)
String fields	Atomic	MUST canonicalize	Unicode normal form per D.6 (e.g., NFC)

D.12 Error List Treatment (Normative)

D.12.1 Terminology

An “Error List” is any structure that contains multiple error objects (e.g., errors, error_stack, error_list), regardless of whether it occurs in the wire envelope or inside payload structures.

D.12.2 Semantics of the Error List

Unless explicitly specified otherwise, an Error List is interpreted as a set in this standard context. This means:

- the order of error objects is not semantic,
- implementations may emit errors in any order,
- for conformance and Golden Outputs, a canonical order MUST be produced.

D.12.3 Normalizer Rule (Canonical Ordering)

The output normalizer MUST canonically sort all Error Lists.

Sort key (total order): error objects MUST be ordered using a total key:

1. primary: error_code (lexicographic)
2. secondary: error_class (lexicographic; if present)
3. tertiary: location/path/field (lexicographic; if present)
4. quaternary: deterministic normalized serialization of the full error object per D.3–D.8

If a field is absent, it is treated as the empty string for sorting.

D.12.4 Duplicates

Duplicates are permitted. The normalizer MUST NOT remove duplicates. Sorting is stable under the full sort key (including the quaternary key), so identical error objects are ordered deterministically.

D.12.5 Consequence for Expected Outputs (Golden Outputs)

If a conformance test defines an expected output, the Error List in that expected output MUST already be in normalized canonical order.

D.12.6 Exception: Semantically Ordered Error Lists

Only if a document in the standard family explicitly specifies that an Error List is an ordered sequence (e.g., “error_stack is ordered by time of occurrence”) then:

- the normalizer MUST preserve order, and
- the structure MUST be explicitly listed as order-significant in the Sorting Matrix.

E) Conformance Harness README

Conformance Suite Structure

Test cases are stored as directories: tests/<Lk>/<TEST-ID>/ with at least:

- input/ (test input),
- expected/ (expected output),
- meta.json (metadata).

Minimal content of meta.json:

meta.json MUST contain:

- test_id,
- level (L1–L4),
- expected_outcome (PASS|FAIL),
- references to normative document locations (document + text anchor).

Normative status of expected outputs:

1. expected/ defines the normative reference output of the test case (“Golden Output”).
2. Comparison is performed against the normalized output per section D.

Purpose

The harness executes conformance test vectors and compares normalized outputs with goldens.

Adapter Contract

An implementation provides a module:

- run_case(case: dict) -> dict
  Return value: JSON-serializable dict (wire envelope).

Notes on L3/L4

For L3/L4 tests, run_case additionally must:

- provide trajectory outputs (trajectory + step logs),
- provide meta outputs (observation + theta progression).

Execution

1. Write goldens:
   (i) python tests/harness/runner.py --adapter <MODULE> --update-goldens
2. Check against goldens:
   (i) python tests/harness/runner.py --adapter <MODULE>

E.1 Golden Test Vector Package

tests/<Lk>/<TEST-ID>/

- meta.json
- input/
- expected/output.json

Normative rules:

- For each test in catalog C, a directory tests/<Lk>/<TEST-ID>/ MUST exist.
- expected/output.json is the normative reference output.
- Comparison uses the normalized output (“byte-identical normalized output”).

E.2 meta.json Template (minimal, normative)

```
{
  "test_id": "<TEST-ID>",
  "level": "<L1|L2|L3|L4>",
  "reqs": ["<REQ-....>", "..."],
  "expected_outcome": "PASS",
  "normative_refs": [
    {
      "doc": "MLD_INTEROP_STD_v1.0",
```

```
    "anchor": "<literal test-catalog entry from C: TEST-ID + Check: ...>"
   }
  ]
}
```


E.3 test-index

test_id	level	reqs	check
T-WIRE-001	L1	REQ-0101	Output satisfies Wire Schema (Envelope)
T-WIRE-002	L1	REQ-0102	Wire sequences are canonically ordered
T-WIRE-003	L1	REQ-0103	Error codes are mapped correctly
T-DET-001	L2	REQ-0001	Identical input ⇒ byte-identical normalized output
T-ORD-001	L2	REQ-0002	Dimensions appear in DimOrder
T-ORD-002	L2	REQ-0003	Candidate/Reach lists are CanonKey-sorted
T-ORD-003	L2	REQ-0004	Maps are deterministically serialized
T-CAN-001	L2	REQ-0005	key_eqclass/CANON is stable
T-ERR-001	L2	REQ-0006	Partial functions return Result[T] (Err instead of hidden fail)
T-DIAG-001	L2	REQ-0007	State is ordinal; errors/confidence separate (no state downgrade)
T-COV-001	L2	REQ-0008	Img_i_impl is non-empty
T-COV-002	L2	REQ-0009	$\varphi \in Img_i$_impl
T-COV-003	L2	REQ-0010	NoOp exists and Apply(NoOp)=Ok(phi)
T-DISC-001	L2	REQ-0011	bind_params/horizon/snapshot/fingerprints present
T-ADAPT-001	L3	REQ-0201	Each update step satisfies non-degradation + at least one strict improvement
T-ADAPT-002	L3	REQ-0202	Output contains no aggregated profile number / no total-order encoding; selection uses Pareto + tie-break
T-ADAPT-003	L3	REQ-0203	Selection protocol shows correct 3-stage selection (Pareto-max → EditCost → CanonKey)
T-ADAPT-004	L3	REQ-0204	CanonKey is not used before EditCost
T-ADAPT-005	L3	REQ-0205	If U is empty ⇒ result = phi
T-ADAPT-006	L3	REQ-0206	Fixpoint/cycle/MaxSteps are detected correctly
T-ADAPT-007	L3	REQ-0207	Logs contain at least phi_n, i_n, phi_n1, H, admissible_count / progress justification
T-META-001	L4	REQ-0301	ThetaImpl manifest + Theta0 present
T-META-002	L4	REQ-0302	Observe(trajectory, logs) is deterministic
T-META-003	L4	REQ-0303	NextTheta is deterministic (Result scheme)
T-META-004	L4	REQ-0304	Schema switching only if RankTheta does not increase (per spec)