MODIVX STANDARD DEFINITION IMPLEMENTATION PROFILE

Protocol version	1.0
Document version:	1.0
Status:	Release Candidate
Category:	Standards Track
Date:	2026-01
Author:	© 2026 Ruben Jaybird Institute

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 (Tests and Certification)

Status of This Document (Runtime Rules)

The Implementation Profile specifies the concrete execution logic and behavioral requirements that ensure a robust, deterministic, and reproducible implementation Rationale for the late layer positioning: This is an operationalization of the preceding definitions; implementation decisions must not modify the fundamental meanings and rules but must strictly adhere to them.

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

1. Scope and Goals

1.1 Scope

This Implementation Profile standardizes the internal behavior of an analyzer implementing the MLD-SDP protocol, including:

- mandatory processing steps per protocol state (S1–S6),
- deterministic ordering and canonicalization rules,
- normative mappings from internal failures to protocol ERROR responses,
- conformance levels for staged interoperability.

1.2 Intended Outcomes

Implementations conforming to this document MUST:

- produce wire messages consistent with the Wire Specification,
- exhibit deterministic behavior under identical inputs (at the appropriate conformance level),
- map internal failures to standard ERROR codes/phases.

2. Conformance Levels

This document defines four conformance levels. An implementation MUST declare at least one supported level.

2.1 Level L1 — Wire Interoperability

L1 implementations MUST:

- parse and emit all wire messages correctly,
- follow the protocol state machine,
- implement required error handling (exactly one ERROR on FAILED).

2.2 Level L2 — Deterministic Findings

L2 implementations MUST satisfy L1 and MUST:

- implement canonical ordering for all repeated structures,
- ensure deterministic exploration/classification outputs given identical inputs and registry snapshot.

2.3 Level L3 — Adaptive Compliance

L3 implementations MUST satisfy L2 and MUST:

- implement adaptive update behavior consistent with the Mathematical Core constraints:
  - → progress condition enforcement,
  - → stabilization guarantees within configured bounds.

2.4 Level L4 — Meta Compliance

L4 implementations MUST satisfy L3 and MUST:

- implement META_DIAGNOSTIC_FINDING generation and meta-stability evaluation per the protocol specification.

3. Reference Model and Terminology

3.1 Roles

- Client: sends DIAGNOSE_REQUEST, receives results.
- Analyzer: processes requests, emits findings and errors.
- Registry: authoritative source of IDs and specifications (dimension set, transforms, context definitions, etc.).

3.2 Execution Context

A diagnostic run MUST be associated with:

- an immutable Protocol Version,
- an immutable Registry Snapshot (see §4.4),
- one purpose context parameter,
- one dimension index set.

3.3 Protocol states

The Analyzer MUST implement the following protocol states:

- S0 IDLE
- S1 REQUESTED
- S2 INITIALIZED
- S3 EXPLORING
- S4 CLASSIFYING
- S4A ADAPTING
- S5 COMPLETED
- S6 FAILED

3.4 Wire messages

1. DIAGNOSE_REQUEST
2. DIAGNOSTIC_FINDING
3. META_DIAGNOSTIC_FINDING
4. ERROR

3.5 Error phases

Every ERROR emitted by the Analyzer MUST include an error phase:

- VALIDATING
- EXPLORING
- EVALUATING
- ADAPTING
- GENERAL

4. Determinism and Canonicalization

4.1 Determinism Requirement

For conformance level L2 and above, the Analyzer MUST be deterministic:

Given the same DIAGNOSE_REQUEST, the same registry snapshot, and the same initial carrier state, the Analyzer MUST produce byte-for-byte identical DIAGNOSTIC_FINDING content after canonicalization (see §4.2).

Definition (DimOrder): DimOrder ist the deterministic dimension iteration order used for emitting dimension-scoped results/findings. DimOrder MUST contain each dimension exactly once. Canonical derivation of DimOrder is the list of all dimensions sorted lexicographically by normalized dimension_id.

4.2 Canonical Ordering Rules

Where the wire format allows repetition (lists/sets), the Analyzer MUST apply stable ordering:

1. Dimension blocks MUST be ordered according to BindParams.dim_order (DimOrder). If dim_order is not explicitly provided, the Analyzer MUST derive dim_order as the canonical dimension identifier order (lixicographic by normalized dimension_id) and disclose it in bind_params.dim_order.
2. Transform blocks MUST be ordered by transform identifier order as received, unless the protocol explicitly defines an order; if multiple transforms commute, the Analyzer MUST use a deterministic tie-breaker. Tie-breakers

and deterministic selection rules MUST NOT be interpreted as semantic rankings of diagnostic profiles; they exist solely to ensure reproducible deterministic behavior.

3. Carrier variants MUST be ordered deterministically by:
   - carrier_equivalence_id,
   - transform sequence identifier,
   - canonical hash of normalized carrier representation.

4.3 Canonical Representation

The Analyzer MUST define and use a canonical representation for:

- carrier representations relevant to exploration,
- context fingerprints,
- dimension identifiers,
- token normalization (case, whitespace, encoding).

4.4 Registry Snapshot Binding

The Analyzer MUST bind each diagnostic run to an immutable registry snapshot:

- either by explicit snapshot identifier, or
- by caching rules that ensure all referenced IDs resolve consistently throughout the run.

If registry data changes mid-run, the Analyzer MUST treat this as an error condition unless the wire specification explicitly permits dynamic registry reads. Where the Wire specification defines a Disclosures object, binding information MUST be wire-visible, at a minimum by disclosing registry_snapshot_id together with the comparability fingerprints.

4.5 Transform Ordering Semantics

This section eliminates ambiguity regarding transform ordering and ensures deterministic behavior across implementations.

4.5.1 Transform Container Type

For each run, the Analyzer MUST classify the request's transform specification as exactly one of:

- TRANSFORM-SEQUENCE (order-sensitive)
- TRANSFORM-SET (order-insensitive)

This classification MUST be deterministic and MUST be based solely on the wire request semantics defined by the Wire Specification.

4.5.2 TRANSFORM-SEQUENCE (Order-Sensitive)

If the transform specification is a TRANSFORM-SEQUENCE:

- The Analyzer MUST preserve the transform order exactly as received on wire.
- The Analyzer MUST NOT reorder transforms for canonicalization purposes.
- Canonicalization MAY normalize transform identifiers and metadata, but MUST NOT change order.

Determinism requirement (L2+):

Given identical request order and identical registry snapshot, transform application order MUST be identical.

4.5.3 TRANSFORM-SET (Order-Insensitive)

If the transform specification is a TRANSFORM-SET:

The Analyzer MUST apply canonical ordering before use:

- order transforms by normalized transform identifier (lexicographic ascending).

The Analyzer MUST treat the request order as non-semantic.

Tie-breaker (required):

- If two transforms have equal ordering keys after normalization, the Analyzer MUST break ties by full canonical identifier bytes (UTF-8).

4.5.4 Transform Commutativity

If transforms are mathematically commutative but encoded as TRANSFORM-SEQUENCE:

- The Analyzer MUST still preserve the wire order (do not reorder).
- Commutativity MUST NOT be used as an excuse to change the observable pipeline.

5. Internal Module Requirements (Reference Architecture)

A conforming Analyzer MUST implement the following logical modules (they may be co-located in code):

- Parser: wire message parsing and ABNF validation
- Validator: semantic validation against registry snapshot
- Explorer: generates and prunes carrier variants

- Classifier: computes dimension-wise diagnostic outputs
- Adaptive Engine: applies update operators and enforces progress constraints
- Emitter: canonicalizes and emits wire messages
- Error Mapper: maps internal failures to protocol ERROR codes and phases

6. Algorithm Profiles by Protocol State

This section is normative. For each protocol state, it defines REQUIRED processing steps, failure conditions, and outputs.

6.1 Global invariants (apply to all states)

Invariant G1 — Single terminal outcome

- Each run MUST terminate in exactly one terminal state: S5 COMPLETED or S6 FAILED.

Invariant G2 — ERROR emission rule

- If and only if the Analyzer transitions into S6 FAILED, it MUST emit exactly one ERROR message.
- The terminal failure MUST be encoded on the wire as WireEnvelope.type=”ERROR”

Invariant G3 — Deterministic scheduling

- For L2+, all loops and traversals MUST be executed in a deterministic order defined by §4.

6.2 S0 — IDLE

Entry conditions: No active run.

Transition rule: Upon receiving a DIAGNOSE_REQUEST, transition to S1 REQUESTED.

Pseudocode

```
STATE S0_IDLE:
  wait for DIAGNOSE_REQUEST
  on receive request:
    set run_context
    goto S1_REQUESTED
```

6.3 S1 — REQUESTED (Validating phase)

Inputs: DIAGNOSE_REQUEST

Required steps:

- Parse request headers and body.
- Validate required fields are present.
- Assign processing context: protocol version, job-id (if present), request timestamp.

Failure conditions:

- malformed request syntax,
- missing required fields,
- unsupported version.

On failure: MUST transition to FAILED and emit ERROR.

On success: MUST transition to S2 INITIALIZED.

Pseudocode:

```
STATE S1_REQUESTED:
  parse request using ABNF
  if ABNF invalid:
    FAIL(code=MALFORMED_REQUEST, phase=VALIDATING)

  if required fields missing:
    FAIL(code=MISSING_REQUIRED_FIELD, phase=VALIDATING)

  if protocol version unsupported:
    FAIL(code=UNSUPPORTED_VERSION, phase=VALIDATING)

  goto S2_INITIALIZED
```

6.4 S2 — INITIALIZED (Validating phase)

Inputs: validated request

Required steps:

- Resolve registry references (dimension_set_id, context, transforms).
- Verify compatibility constraints between resolved entities.
- Confirm at least one permissible transformation pipeline exists.

Failure conditions:

- unknown references,
- inconsistent context/dimension set linkage,
- no valid transformation pipeline.

On success: MUST transition to S3 EXPLORING.

Pseudocode:

```
STATE S2_INITIALIZED:
  bind registry_snapshot

  resolve dimension_set_id
  resolve context definition
  resolve transform definitions

  if any reference unresolved:
    FAIL(code=UNKNOWN_REFERENCE, phase=VALIDATING)

  if compatibility constraints violated (context vs dimension set vs transforms):
    FAIL(code=CONTEXT_INCONSISTENT, phase=VALIDATING)

  if no permissible transform pipeline exists:
    FAIL(code=NO_VALID_TRANSFORM_PIPELINE, phase=VALIDATING)

  goto S3_EXPLORING
```

6.5 S3 — EXPLORING (Exploring phase)

Inputs: initialized exploration configuration

Required steps:

- Initialize exploration frontier deterministically.
- Repeatedly apply transforms to generate candidate variants.
- Apply pruning rules deterministically.
- Terminate exploration when completion criteria are met.

Loop requirement:

- Exploration MUST be implemented as an explicit iteration until completion criteria are satisfied or exploration becomes impossible.

Completion criteria:

- the Analyzer has enumerated all required candidates under the selected exploration policy, OR
- a protocol-defined completion threshold is met (if configured deterministically).

Failure conditions:

- exploration cannot proceed,
- exploration violates configured bounds (see §7.2),
- internal inconsistency.

On success: MUST transition to S4 CLASSIFYING with a deterministically ordered candidate set.

Pseudocode:

```
STATE S3_EXPLORING:
  init frontier deterministically
  init explored_set deterministically

  while NOT exploration_completed:
    select next node from frontier deterministically
    if frontier empty:
      FAIL(code=EXPLORATION_EMPTY, phase=EXPLORING)

    expand node using transforms in canonical order
    for each candidate variant:
      if invalid variant:
        continue
      canonicalize variant representation
      if variant not in explored_set:
        add to explored_set
        add to frontier deterministically

    if bounds violated:
      FAIL(code=EXPLORATION_BOUND_EXCEEDED, phase=EXPLORING)

  build ordered_candidate_set from explored_set (canonical order)
  goto S4_CLASSIFYING
```

6.5.1 Exploration Completion Policy

6.5.1.1 Core Rule: Frontier Exhaustion Under Bounds

Exploration MUST be considered COMPLETED if and only if:

- the exploration frontier becomes empty, OR
- the Analyzer reaches a configured bound that forces termination, AND the selected completion mode permits completion.

6.5.1.2 Candidate Enumeration Policy

The Analyzer MUST implement candidate enumeration as follows:

- Initialize frontier with exactly one initial node derived from the request carrier input.
- Execute the loop:
    - select next node deterministically (Annex C),
    - expand by applying transforms (per §4.5),
    - canonicalize the candidate representation,
    - deduplicate candidates using canonical keys only,

    - insert new nodes into frontier deterministically.
- Exploration output set MUST be the set of all deduplicated nodes processed or discovered before completion.

6.5.1.3 Completion Mode (Interop Lock)

The Analyzer MUST support exactly the following completion modes:

- MODE-FULL (default):
  Exploration MUST run until frontier exhaustion (empty frontier).
  If a bound is exceeded, the run MUST FAIL unless allow_partial permits partial completion.
- MODE-BOUNDED (optional):
  Exploration MAY stop when a preconfigured maximum is reached, but only if:
    - allow_partial is enabled (§8.3), AND
    - output marking rules for partial results are satisfied (§8.3.4).

Conformance rule:

- Implementations claiming L2+ MUST declare which completion modes they support.
- MODE-FULL MUST be supported by all implementations.

6.5.1.4 Deterministic Output Requirement (L2+)

For L2+:

- the final ordered candidate set MUST be derived by canonical ordering (Annex B),
- and MUST be identical across runs with identical inputs and snapshot.

6.6 S4 — CLASSIFYING (Evaluating phase)

Inputs: exploration results

Required steps:

- For each candidate and each dimension:
    - compute diagnostic dimension outputs as specified by the mathematical core assignment system,
    - record classification outcomes in deterministic order.
- Determine run-level stability/equivalence fields per wire specification.

Failure conditions:

- classification undefined or inconsistent,
- numerical instability beyond configured tolerance (if applicable).

On success:

MUST either:

- transition to S5 COMPLETED, OR
- transition to S4A ADAPTING if adaptive execution is active.

Pseudocode:

```
STATE S4_CLASSIFYING:
  for each candidate in ordered_candidate_set:
    for each dimension in canonical_dimension_order:
      compute diagnostic outcome for dimension
      if outcome undefined / inconsistent:
        FAIL(code=CLASSIFICATION_UNDEFINED, phase=EVALUATING)
      record result deterministically

  if adaptive execution active:
    goto S4A_ADAPTING
  else:
    goto S5_COMPLETED
```

6.6.1 Candidate ↔ Equivalence Class Mapping

6.6.1.1 Definitions Candidate ↔ Equivalence Class Mapping

- A candidate is an explored carrier variant produced during S3.
- A carrier equivalence id is the canonical identifier used to group candidates into equivalence classes.

6.6.1.2 Normative Mapping Rule

For protocol execution purposes:

- Each candidate MUST be associated with exactly one equivalence class identifier:
  carrier_equivalence_id.

The Analyzer MUST interpret:

- one equivalence class [x] ⇔ all candidates sharing the same carrier_equivalence_id.

6.6.1.3 Representative Selection (Required)

When the core semantics require selecting a representative [x] for evaluation:

- The Analyzer MUST select a deterministic representative candidate per equivalence class: the candidate with the smallest candidate_key (Annex B).

This representative MUST be used consistently for:

- dimension evaluation where only a single representative is allowed,

- aggregation baselines.

6.6.2 Aggregation Rules per Dimension

6.6.2.1 Dimension Aggregation Type

For each dimension, the Analyzer MUST determine one aggregation type from registry metadata (preferred) or an implementation-declared profile (fallback):

- AGG-REP: Representative-only evaluation
- AGG-ALL: Evaluate all candidates within class and aggregate
- AGG-ANY: Evaluate until first decisive result
- AGG-MAJORITY: Majority vote across candidates (only if dimension supports voting)

6.6.2.2 Required Default

If no aggregation type is specified by registry metadata:

The Analyzer MUST default to AGG-REP.

6.6.2.3 Normative Rules per Aggregation Type

AGG-REP

- Evaluate only the representative candidate selected per §6.6.1.3.

AGG-ALL

- Evaluate all candidates within the class in canonical candidate order.
- Aggregate results using a deterministic dimension-defined aggregation operator.
- If dimension does not define aggregation operator: MUST FAIL with CLASSIFICATION_UNDEFINED.

AGG-ANY

- Evaluate candidates in canonical order.
- Stop at the first candidate yielding a decisive result.
- If no candidate yields a decisive result: apply dimension-defined fallback, else FAIL.

AGG-MAJORITY

- Evaluate all candidates.
- Compute majority across normalized outcome tokens.
- Tie-break MUST be deterministic by canonical outcome token order.

6.6.2.4 Required Reporting

The Analyzer MUST record in DIAGNOSTIC_FINDING, per dimension:

- aggregation type applied,
- candidate count considered for the class (if relevant).

6.7 S4A — ADAPTING (Adapting phase)

Inputs: classification state vector(s)

Required steps:

- Apply the adaptive update operator.
- Enforce progress constraints after each update.
- Iterate until stabilization or configured bound is reached.

Loop requirement:

- Adaptation MUST be modeled as an explicit iteration (update loop).
- For L3+, the loop MUST enforce progress constraints.

Failure conditions:

- update operator cannot be applied,
- progress violation,
- failure to stabilize within bounds.

On success: MUST transition to S5 COMPLETED.

Pseudocode:

```
STATE S4A_ADAPTING:
  init adaptive_state from classification_state

  iter = 0
  while NOT stabilized:
    if iter >= MAX_ADAPT_ITERS:
      FAIL(code=ADAPTATION_BOUND_EXCEEDED, phase=ADAPTING)

    next_state = apply_update_operator(adaptive_state)
    if update failed:
      FAIL(code=UPDATE_FAILED, phase=ADAPTING)

    if progress violated:
      FAIL(code=PROGRESS_VIOLATION, phase=ADAPTING)

    adaptive_state = next_state
    iter += 1

  finalize adaptive_state
  goto S5_COMPLETED
```

6.8 S5 — COMPLETED

Required steps:

- Construct a DIAGNOSTIC_FINDING message.
- Apply canonical ordering (§4).
- Emit DIAGNOSTIC_FINDING.

Optional step:

MAY emit META_DIAGNOSTIC_FINDING if supported and allowed by the wire specification.

Pseudocode:

```
STATE S5_COMPLETED:
  construct DIAGNOSTIC_FINDING from run_state
  canonicalize output (ordering rules §4)
  emit DIAGNOSTIC_FINDING

  if meta enabled and permitted:
    emit META_DIAGNOSTIC_FINDING

  terminate run
```

6.9 S6 — FAILED

Required steps:

- Map failure to ERROR code and phase.
- Emit exactly one ERROR message. The emitted ERROR MUST be encoded as WireEnvelope.type=”ERROR”.

Optional step:

MAY emit META_DIAGNOSTIC_FINDING after ERROR if supported.

Pseudocode:

```
STATE S6_FAILED:
  emit exactly one ERROR (code + phase + required metadata)
  if meta enabled and permitted:
    emit META_DIAGNOSTIC_FINDING
  terminate run
```

7. Bounds, Limits, and Termination

7.1 Mandatory Bounds

The Analyzer MUST define explicit operational bounds for:

- maximum exploration nodes (MAX_EXPLORATION_NODES)
- maximum transform depth (MAX_TRANSFORM_DEPTH)
- maximum adaptation iterations (MAX_ADAPT_ITERS)
- maximum per-run wall time (MAX_WALL_TIME)

7.2 Bound Violations

When a bound is violated:

L1: MAY return an implementation-defined error code if the wire spec permits.

L2+: MUST map to a deterministic ERROR code/phase.

7.3 Partial Result Policy (allow_partial)

7.3.1 Meaning of allow_partial

If allow_partial = false (default):

- The Analyzer MUST NOT emit a completed finding unless all required phases and required dimension outputs have been produced.
- Any failure MUST transition to FAILED and emit ERROR.

If allow_partial = true:

- The Analyzer MAY emit a completed finding with partial results under strict conditions defined below.

7.3.2 Partial Completion Conditions

A run MAY complete partially if and only if ALL conditions hold:

1. The run reached a point where at least one dimension output exists for at least one equivalence class.
2. The failure preventing full completion is one of:
    - EXPLORATION_BOUND_EXCEEDED
    - ADAPTATION_BOUND_EXCEEDED
3. a dimension-specific non-critical failure explicitly marked as “partial-eligible” by registry metadata.
4. The Analyzer can produce a deterministic set of outputs consistent with §4.

If the failure is not in the eligible set, the Analyzer MUST FAIL even if allow_partial is true.

7.3.3 Required Marking in DIAGNOSTIC_FINDING

If the Analyzer emits a partial DIAGNOSTIC_FINDING, it MUST include:

- partial_result_flag = true

- partial_reason_code (deterministic token)
- missing_dimension_ids[..] (canonical order)
- missing_equivalence_class_ids[..] (canonical order), if applicable
- bound_counters (nodes explored, depth, iterations)

7.3.4 Deterministic Partial Semantics (L2+)

For L2+:

- the set of included dimension results MUST be deterministic,
- the set of missing dimensions/classes MUST be deterministic,
- the partial_reason_code MUST be deterministic and stable across runs.

7.3.5 Partial vs ERROR Exclusivity

If a partial DIAGNOSTIC_FINDING is emitted:

- the Analyzer MUST NOT transition to FAILED,
- and MUST NOT emit ERROR for that run.

8. Error Mapping Profile

8.1 Error Phase Assignment

The ERROR message MUST include an error phase consistent with the failing state:

VALIDATING → S1/S2

EXPLORING → S3

EVALUATING → S4

ADAPTING → S4A

GENERAL → internal/systemic

8.2 Internal_Failure-to-ERROR Mapping Table

Implementations MUST maintain a mapping table:

- internal_failure → (ERROR.code, ERROR.phase)

This mapping MUST be deterministic.

9. Conformance Checklist

An implementation claiming conformance to level Lx MUST:

- satisfy all MUST requirements for that level and all lower levels,

- pass the corresponding conformance test suite (Document B).

10. Internal Data Model (Required for L2+)

10.1 Purpose

This section defines a normative internal data model. Implementations MAY choose their own programming language and storage strategy, but MUST preserve the semantics, invariants, and determinism requirements defined here.

10.2 Core internal objects

A conforming Analyzer MUST represent each diagnostic run using the following logical objects.

10.2.1 RunContext

Fields (REQUIRED)

- protocol_version
- job_id (if present on wire; otherwise generated)
- request_time
- registry_snapshot_id (or equivalent immutability marker)
- conformance_level (L1–L4)
- bounds (see §7.1)

Invariants (MUST)

- RunContext MUST be immutable after S2 INITIALIZED.
- Registry snapshot binding MUST remain stable for the full run (§4.4).

10.2.2 RequestModel

Represents normalized DIAGNOSE_REQUEST content.

Fields (REQUIRED)

- dimension_set_id
- context_reference
- transform_spec
- allow_partial

Invariants (MUST)

- RequestModel MUST be canonically normalized (identifier normalization, whitespace, encoding).
- Any decoding ambiguity MUST result in FAIL at VALIDATING phase.

10.2.3 ResolvedRegistryModel

Represents all resolved registry entities used by the run.

Fields (REQUIRED)

- dimension_set (resolved)
- context_definition (resolved)
- transform_definitions[] (resolved, ordered)
- compatibility_constraints

Invariants (MUST)

- Every resolved entity MUST be derived from the bound snapshot.
- The ordering of transform_definitions[] MUST be deterministic (§4.2).

10.2.4 ExplorationNode

Represents a single exploration state/variant candidate.

Fields (REQUIRED)

- carrier_equivalence_id (or canonical equivalent)
- transform_sequence_id
- canonical_carrier_hash
- normalized_carrier_representation

Invariants (MUST)

- Canonical hash MUST be stable across runs at L2+.
- Equality of nodes MUST be determined solely by canonical fields.

10.2.5 ExplorationFrontier

Represents the exploration working set.

Fields (REQUIRED)

- frontier (ordered container)
- explored_set (deduplication set)
- candidate_set (final output set)

Invariants (MUST)

- Frontier selection order MUST be deterministic at L2+.
- Candidate_set ordering MUST follow §4.2.

10.2.6 ClassificationResult

Represents classification output for one candidate and one dimension.

Fields (REQUIRED)

- candidate_id
- dimension_id
- classification_payload (dimension-defined)
- valid_flag (protocol validity flag, if applicable)
- diagnostic_metadata

Invariants (MUST)

- dimension_id MUST be normalized.
- Repeated computation for same (candidate, dimension) MUST be stable at L2+.

10.2.7 ClassificationState

Represents the full classification output state.

Fields (REQUIRED)

- results[] (ordered list)
- dimension_index (fast lookup)
- run_level_properties (stability/equivalence metadata)

Invariants (MUST)

- results[] MUST be emitted in canonical dimension order and canonical candidate order at L2+.

10.2.8 AdaptiveState (L3+)

Represents the adaptive dynamics state.

Fields (REQUIRED)

- current_state_vector
- iteration_index
- progress_witness (data supporting progress checks)
- stabilized_flag

Invariants (MUST)

- Each update step MUST be progress-checked (L3+).
- Stabilization decision MUST be deterministic under identical inputs.

10.2.9 EmissionModel

Represents the final outgoing wire message structure.

Fields (REQUIRED)

- diagnostic_finding
- optional_meta_finding
- optional_error

Invariants (MUST)

Exactly one of:

- diagnostic_finding OR error
- MUST be produced per run.

Canonical ordering MUST be applied before serialization.

10.3 Deterministic identifiers (Normative)

For L2+ the Analyzer MUST define deterministic, canonical identifiers:

- candidate_id
- transform_sequence_id
- canonical hash algorithm and canonical input serialization

If a cryptographic hash is used, the algorithm MUST be fixed per protocol version.

10.4 Numeric stability profile

If classification relies on floating-point evaluation, the Analyzer SHOULD:

- define rounding mode and tolerance,
- fail deterministically when tolerance is exceeded,
- avoid platform-dependent non-determinism.

11. Observability Requirements (L2+)

To enable debugging and certification, the Analyzer MUST be able to produce an execution trace (not necessarily over wire) containing:

- state transitions S0→…→S5/S6,
- bound counters (nodes explored, adaptation iterations),
- canonicalization decisions (ordering keys),
- error mapping (internal failure → ERROR code/phase).

12. Security and Robustness Profile

12.1 Purpose

This section defines mandatory robustness and security requirements for analyzers implementing MLD-SDP. These requirements are intended to:

- prevent malformed input exploitation,
- ensure predictable resource usage,
- avoid information leakage through ERROR metadata,
- mitigate denial-of-service patterns in exploration/adaptation loops.

12.2 Input Validation and Parsing Hardening

12.2.1 Strict ABNF enforcement

The Analyzer MUST apply strict ABNF validation before any semantic processing.

The Analyzer MUST NOT attempt “best-effort” parsing under L2+.

12.2.2 Maximum input sizes

The Analyzer MUST enforce explicit limits:

- maximum request size (bytes),
- maximum header size,
- maximum number of repeated elements (e.g., transform entries).

Violations MUST fail deterministically with:

- ERROR.phase = VALIDATING
- ERROR.code = MALFORMED_REQUEST or BOUND_EXCEEDED profile code (see Annex A), depending on local mapping policy.

12.3 Registry Safety

12.3.1 Snapshot integrity

Each run MUST bind to an immutable registry snapshot (§4.4).

If snapshot integrity cannot be established, the Analyzer MUST fail:

- ERROR.phase = GENERAL
- ERROR.code = INTERNAL_ERROR

12.3.2 Registry lookup bounds

The Analyzer MUST enforce limits on:

- maximum registry lookups per run,
- maximum recursion depth for registry references (if applicable),
- maximum resolution time.

Bound violations MUST fail deterministically (L2+), with:

- ERROR.phase = VALIDATING
- ERROR.code = UNKNOWN_REFERENCE or INTERNAL_ERROR (per Annex A mapping policy).

12.4 Exploration DoS Mitigation (S3)

12.4.1 Exploration bounds are mandatory

For all conformance levels, the Analyzer MUST enforce:

- MAX_EXPLORATION_NODES,
- MAX_TRANSFORM_DEPTH,
- MAX_WALL_TIME.

12.4.2 Cost-based pruning (REQUIRED for L3+ deployments)

The Analyzer SHOULD implement cost heuristics for pruning exploration candidates.

If implemented, pruning MUST be deterministic (L2+) and MUST NOT affect conformance semantics (only performance).

12.4.3 Non-termination protection

The Analyzer MUST guarantee that exploration terminates via explicit bounds.

If bounds are reached:

- ERROR.phase = EXPLORING
- ERROR.code = EXPLORATION_BOUND_EXCEEDED

12.5 Adaptive Loop Safety (S4A)

12.5.1 Adaptation bounds

The Analyzer MUST enforce:

- MAX_ADAPT_ITERS,
- MAX_WALL_TIME.

12.5.2 Progress enforcement

For L3+:

The Analyzer MUST check progress constraints after every update step.

A progress violation MUST fail deterministically with:

- ERROR.phase = ADAPTING
- ERROR.code = PROGRESS_VIOLATION

12.6 ERROR Metadata Sanitization

12.6.1 No sensitive leakage

The ERROR message MUST NOT include:

- memory addresses,
- stack traces,
- internal file paths,
- raw registry contents,
- raw carrier payloads, unless explicitly permitted by the wire spec.

12.6.2 Correlation identifiers

The Analyzer SHOULD include a correlation identifier (job id / trace id), if permitted by the wire spec.

12.7 Replay and Audit

For L2+:

The Analyzer MUST support reproducibility under a fixed registry snapshot.

Implementations MUST be able to produce an audit trace (see §11) sufficient to reproduce:

- state transitions,
- exploration ordering,
- bound counters,
- error mapping decisions.

13. Extension Points and Compatibility Rules

13.1 Purpose

This section defines how implementations MAY extend the system (new dimensions, transforms, metadata) while preserving:

- wire interoperability,
- determinism guarantees (L2+),
- adaptive compliance (L3+),
- forward/backward compatibility across protocol versions.

13.2 Extension Point Categories

Extensions MUST fall into one of the following categories.

13.2.1 Registry Extensions

Extensions introduced via registry entries, including:

- new dimension_set entries,
- new context definitions,
- new transform definitions.

Rule: Registry extensions MUST be snapshot-consistent (§4.4) and MUST NOT change semantics of existing identifiers.

13.2.2 Optional Wire Fields / Metadata

If the wire specification allows optional fields or metadata blocks:

An Analyzer MAY include them.

A Client MUST ignore unknown optional fields unless the wire specification states otherwise.

Rule: Unknown optional fields MUST NOT change the interpretation of required fields.

13.2.3 Implementation-Internal Heuristics

Implementations MAY use private heuristics (e.g., pruning strategies), provided:

- L2+ determinism is preserved for observable outputs,
- conformance tests still pass.

13.3 Versioning Rules

13.3.1 Protocol Version Compatibility

An Analyzer MUST declare supported protocol versions.

- If request protocol version is unsupported:
- MUST fail with ERROR.phase = VALIDATING
- MUST use ERROR.code = UNSUPPORTED_VERSION

13.3.2 Backward Compatibility

For a supported version v, the Analyzer MUST:

- accept all valid messages that conform to v,
- preserve semantics of required fields as defined in v.

13.3.3 Forward Compatibility

If a request contains unknown optional fields:

- Analyzer MUST ignore them (unless explicitly forbidden by the wire spec),
- Analyzer MUST NOT fail solely due to unknown optional fields.

If a request contains unknown required fields (i.e., required by a newer version):

Analyzer MUST fail with UNSUPPORTED_VERSION or equivalent profile code.

13.4 Dimension Extensions

13.4.1 Adding Dimensions

A new diagnostic dimension MAY be added only by:

- defining a new dimension identifier in the registry snapshot,
- including it in a dimension set.

Rule: For L2+ determinism, dimension evaluation order MUST be defined by canonical dimension ordering (§4.2, Annex B).

13.4.2 Dimension Semantics Stability

Once a dimension identifier is published:

its meaning MUST NOT change across minor revisions of registry snapshots.

If semantics must change:

a new dimension identifier MUST be created.

13.5 Transform Extensions

13.5.1 Adding Transforms

A new transform MAY be introduced by registry definition.

Rule: Each transform definition MUST specify:

- its identifier,
- applicable carrier types,
- failure modes,
- whether it is deterministic.

13.5.2 Determinism Requirement for Transforms

For L2+:

- all transforms used in S3 exploration MUST be deterministic.
- if a transform uses randomness:
- it MUST use a deterministic seed derived from RunContext and canonical candidate key, OR
- it MUST be forbidden under L2+.

13.5.3 Transform Ordering and Compositionality

If transforms commute or reorder is possible:

- Analyzer MUST apply a deterministic tie-breaker rule.
- The tie-breaker MUST be documented and MUST be stable across runs.

13.6 Meta Extensions (L4)

If META_DIAGNOSTIC_FINDING is supported:

- unknown meta fields MUST be ignored by consumers,
- the Analyzer MUST ensure meta outputs do not alter DIAGNOSTIC_FINDING semantics.

13.7 Compatibility with the Mathematical Core

For L3+:

- any extension that affects adaptive dynamics MUST preserve:
- progress condition semantics,
- stabilization guarantees within declared bounds.

If this cannot be guaranteed:

the extension MUST be treated as non-conformant for L3+.

14. Compatibility Failure Handling

14.1 Unknown Identifiers

If dimension/context/transform identifiers cannot be resolved within the bound snapshot:

- MUST fail with ERROR.phase = VALIDATING
- MUST use ERROR.code = UNKNOWN_REFERENCE

14.2 Semantic Incompatibility

If resolved entities are incompatible:

- MUST fail with ERROR.phase = VALIDATING
- MUST use ERROR.code = CONTEXT_INCONSISTENT

Annex A — State-to-Error-Code Matrix

A.1 Purpose

This annex defines a normative mapping for common failure conditions. Implementations MUST either:

- implement this mapping exactly, OR
- provide a documented mapping that is behaviorally equivalent and passes conformance tests.

A.2 Matrix

State	Failure condition (normative trigger)	ERROR.phase	ERROR.code (profile)
S1 REQUESTED	ABNF parse failure / malformed structure	VALIDATING	MALFORMED_REQUEST
S1 REQUESTED	missing required fields	VALIDATING	MISSING_REQUIRED_FIELD
S1 REQUESTED	unsupported protocol version	VALIDATING	UNSUPPORTED_VERSION
S2 INITIALIZED	unknown registry reference (dimension/context/transform)	VALIDATING	UNKNOWN_REFERENCE
S2 INITIALIZED	incompatible registry entities	VALIDATING	CONTEXT_INCONSISTENT
S2 INITIALIZED	no permissible transform pipeline	VALIDATING	NO_VALID_TRANSFORM_PIPELINE
S3 EXPLORING	exploration frontier becomes empty prior to completion	EXPLORING	EXPLORATION_EMPTY
S3 EXPLORING	exploration cannot proceed due to internal inconsistency	EXPLORING	EXPLORATION_NOT_FEASIBLE
S3 EXPLORING	bound exceeded (nodes/depth/time)	EXPLORING	EXPLORATION_BOUND_EXCEEDED
S4 CLASSIFYING	classification undefined / inconsistent for required dimension(s)	EVALUATING	CLASSIFICATION_UNDEFINED
S4 CLASSIFYING	numeric instability beyond tolerance (if profile enabled)	EVALUATING	NUMERIC_INSTABILITY
S4A ADAPTING	update operator cannot be applied	ADAPTING	UPDATE_FAILED
S4A ADAPTING	progress condition violation	ADAPTING	PROGRESS_VIOLATION
S4A ADAPTING	bound exceeded (iterations/time)	ADAPTING	ADAPTATION_BOUND_EXCEEDED
any	systemic failure (OOM, corrupted snapshot, invariant violation)	GENERAL	INTERNAL_ERROR

Normative notes

- ERROR.phase MUST match the state where failure is detected (see §8.1).
- For L2+, bound exceedance MUST map deterministically to the specific *_BOUND_EXCEEDED code.
- Exactly one ERROR MUST be emitted upon transition to S6 FAILED.

A.3 Error Code Identifier Binding

A.3.1 Normative Binding Rule

Annex A error codes MUST be interpreted as canonical error identifiers, not examples.

If the Wire Specification defines a different error identifier registry:

- the Analyzer MUST include wire-canonical identifiers in ERROR messages, and
- MAY internally map from Annex A identifiers to wire identifiers, but the emitted ERROR MUST match the wire-defined codes exactly.

A.3.2 Required Implementation Artifact

Each implementation MUST provide a published mapping table:

- AnnexA_code → wire_error_code_id

This table MUST be included in the Conformance Statement (Annex D).

A.3.2 Profile Annex-A Code → Wire Registry Code

Profile ERROR.phase	Profile ERROR.code (Annex A)	Wire error_phase	Wire error_code (Registry)
VALIDATING	MALFORMED_REQUEST	VALIDATING	E1000 MALFORMED_REQUEST
VALIDATING	MISSING_REQUIRED_FIELD	VALIDATING	E1001 MISSING_REQUIRED_FIELD
VALIDATING	UNSUPPORTED_VERSION	VALIDATING	E1002 UNSUPPORTED_VERSION
VALIDATING	UNKNOWN_REFERENCE	VALIDATING	E1003 UNKNOWN_CONTEXT_REFERENCE
VALIDATING	CONTEXT_INCONSISTENT	VALIDATING	E1005 CONTEXT_INCONSISTENT
VALIDATING	NO_VALID_TRANSFORM_PIPELINE	VALIDATING	E1004 INVALID_TRANSFORM_SPEC
EXPLORING	EXPLORATION_EMPTY	EXPLORING	E2005 EXPLORATION_EMPTY
EXPLORING	EXPLORATION_NOT_FEASIBLE	EXPLORING	E2006 EXPLORATION_NOT_FEASIBLE
EXPLORING	EXPLORATION_BOUND_EXCEEDED	EXPLORING	E2004 EXPLORATION_BUDGET_EXCEEDED
EVALUATING	CLASSIFICATION_UNDEFINED	EVALUATING	E3000 CLASSIFICATION_UNDEFINED
EVALUATING	NUMERIC_INSTABILITY	EVALUATING	E3003 INTERNAL_EVALUATION_ERROR
ADAPTING	UPDATE_FAILED	ADAPTING	E4001 UPDATE_NOT_REALIZABLE
ADAPTING	PROGRESS_VIOLATION	ADAPTING	E4002 PROGRESS_VIOLATION
ADAPTING	ADAPTATION_BOUND_EXCEEDED	ADAPTING	E4003 ADAPTION_BUDGET_EXCEEDED
GENERAL	INTERNAL_ERROR	GENERAL	E9000 INTERNAL_ERROR

Annex B — Deterministic Ordering Keys

This annex makes §4.2 directly implementable.

B.1 Dimension ordering key

dimension_key = normalize(dimension_id) (lexicographic ascending)

B.2 Transform ordering key

- if request-ordered list: preserve request order
- else: transform_key = normalize(transform_id) (lexicographic ascending)

B.3 Candidate ordering key

candidate_key = (carrier_equivalence_id, transform_sequence_id, canonical_carrier_hash)

Annex C — Normative Pseudocode Library

C.1 Purpose

This annex defines normative helper algorithms required for L2+ determinism. Implementations MUST produce behavior equivalent to these algorithms.

C.2 Identifier Normalization

C.2.1 normalize_id(id_string)

INPUT: id_string
OUTPUT: normalized identifier string

1) require id_string is valid UTF-8
2) trim leading/trailing whitespace
3) replace internal runs of whitespace with single SPACE
4) map to lowercase ASCII where applicable (A-Z -> a-z)
5) return result

Rule: All ordering keys in §4 and Annex B MUST use normalize_id.

C.3 Canonical Carrier Serialization

C.3.1 canonicalize_carrier(carrier)

INPUT: carrier object (implementation-defined)
OUTPUT: canonical byte sequence

1) serialize carrier into a deterministic structured form:
   - stable field order
   - explicit type tags
   - no non-deterministic metadata (timestamps, pointers)
2) ensure consistent numeric formatting:
   - decimal normalization for floats if present
3) encode as UTF-8 bytes
4) return byte sequence

Rule: The canonical serialization MUST be identical for semantically identical carriers.

C.4 Canonical Hashing Profile

C.4.1 canonical_hash(bytes)

INPUT: byte sequence
OUTPUT: hash string

1) compute HASH_ALGO(bytes)
2) return lowercase hex digest

Rule: HASH_ALGO MUST be fixed per protocol version for L2+.

Implementations MUST declare HASH_ALGO in their conformance statement (Annex D).

C.5 Stable Sorting

C.5.1 stable_sort(list, key_fn)

INPUT: list, key_fn
OUTPUT: sorted list

1) apply stable sort using key_fn
2) if ties remain:
   - preserve original relative order
3) return result

Rule: All lists required to be canonicalized MUST use stable sorting semantics.

C.6 Exploration Frontier Ordering

C.6.1 Frontier node key

For each exploration node, compute:

```
node_key = (
  normalize_id(carrier_equivalence_id),
  normalize_id(transform_sequence_id),
  canonical_carrier_hash
)
```

C.6.2 select_next_node(frontier)

INPUT: frontier: ordered container of nodes
OUTPUT: next node

1) frontier MUST be maintained in ascending node_key order
2) select and remove the smallest node_key element
3) return selected node

Rule: For L2+, the frontier selection order MUST be determined solely by node_key.

C.7 Exploration Candidate Construction

C.7.1 add_candidate(explored_set, frontier, node)

INPUT: explored_set, frontier, node
OUTPUT: updated explored_set, frontier

1) if node.canonical_carrier_hash already in explored_set for same node_key:
    return without modification
2) add node to explored_set
3) insert node into frontier maintaining sorted order by node_key
4) return

Rule: Deduplication MUST be based on canonical fields (§10.2.4).

C.8 Output Canonicalization

C.8.1 Canonical dimension ordering

dimension_key = normalize_id(dimension_id)

C.8.2 Canonical candidate ordering

candidate_key = (carrier_equivalence_id, transform_sequence_id, canonical_carrier_hash)

C.8.3 canonicalize_finding(finding)

INPUT: finding object
OUTPUT: canonicalized finding object

1) stable_sort dimension_blocks by dimension_key
2) within each dimension:
    stable_sort candidate_results by candidate_key
3) stable_sort transform blocks using §4.2 rules
4) return finding

Rule: The emitted DIAGNOSTIC_FINDING MUST be canonicalized before serialization.

Annex D — Conformance Statement Template

D.1 Purpose

Implementations claiming conformance MUST publish a Conformance Statement using this template.

D.2 Template

Implementation Name:

Implementation Version:

Protocol Version(s) Supported:

D.2.1 Conformance Level

Supported level(s):

- L1 Wire Interoperability
- L2 Deterministic Outputs
- L3 Adaptive Compliance
- L4 Meta Compliance

D.2.2 Registry Snapshot Binding

Snapshot binding method:

- explicit snapshot id
- immutable cache policy

Description (required):

D.2.3 Determinism Profile (L2+)

- Identifier normalization: normalize_id equivalent implemented? [ ] yes [ ] no
- Stable sorting semantics implemented? [ ] yes [ ] no
- Canonical carrier serialization method:

  description (required):

- Hash algorithm (HASH_ALGO):
  value (required):

D.2.4 Bounds Configuration

- MAX_EXPLORATION_NODES:
- MAX_TRANSFORM_DEPTH:
- MAX_ADAPT_ITERS:
- MAX_WALL_TIME:

D.2.5 Error Mapping Profile

- Exception-to-ERROR mapping table provided? [ ] yes [ ] no

- Deviations from Annex A mapping (if any): description:

D.2.6 Platform Notes (Optional)

- CPU/OS/runtime:
- Floating-point model and tolerances:

D.2.7 Test Evidence

- Conformance test suite version:
- Date executed:
- Results summary:
- Artifact link or checksum:

Annex E — Minimal Reference Implementation Outline (Informative)

E.1 Purpose

This annex outlines a minimal architecture that can be used by implementers as a blueprint. It is not normative code, but it MUST support all normative requirements of this profile.

E.2 Module Skeleton

E.2.1 Parser

Responsibilities:

- Parse DIAGNOSE_REQUEST per ABNF.
- Construct RequestModel (§10.2.2).
- Reject malformed inputs.

Interfaces:

- parse_request(bytes) -> RequestModel | Failure(VALIDATING)

E.2.2 Semantic Validator

Responsibilities:

- Validate required fields.
- Enforce input bounds (§12.2).
- Pre-validate registry reference formats.

Interfaces:

- validate_request(RequestModel) -> OK | Failure(VALIDATING)

E.2.3 Registry Resolver

Responsibilities:

- Bind registry snapshot.
- Resolve dimension set, context, transforms.
- Provide ResolvedRegistryModel (§10.2.3).

Interfaces:

- bind_snapshot(request) -> snapshot_id
- resolve(snapshot_id, request) -> ResolvedRegistryModel | Failure(VALIDATING)

E.2.4 Explorer

Responsibilities:

- Implement S3 exploration loop deterministically (Annex C.6–C.7).
- Enforce exploration bounds (§7, §12.4).
- Produce ordered candidate set.

Interfaces:

- explore(resolved_model) -> ordered_candidate_set | Failure(EXPLORING)

E.2.5 Classifier

Responsibilities:

- Implement S4 dimension-wise classification.
- Populate ClassificationState (§10.2.7).
- Produce deterministic results (L2+).

Interfaces:

- classify(candidates, resolved_model) -> ClassificationState | Failure(EVALUATING)

E.2.6 Adaptive Engine

Responsibilities:

- Implement S4A update loop.
- Enforce progress constraints (L3+).
- Enforce adaptation bounds (§7, §12.5).

Interfaces:

- adapt(classification_state, resolved_model) -> final_state | Failure(ADAPTING)

E.2.7 Emitter

Responsibilities:

- Construct DIAGNOSTIC_FINDING / META_DIAGNOSTIC_FINDING / ERROR.
- Apply canonicalization (Annex C.8).
- Serialize wire messages.

Interfaces:

- emit_finding(final_state) -> bytes
- emit_error(mapped_error) -> bytes

- emit_meta(meta_state) -> bytes

E.2.8 Error Mapper

Responsibilities:

- Map failures to Annex A matrix.
- Set ERROR.phase based on state.
- Ensure deterministic mappings.

Interfaces:

- map_failure(state, failure) -> (ERROR.code, ERROR.phase, metadata)

E.3 Reference Execution Flow

S0 -> S1:	parse + validate
S2:	bind snapshot + resolve registry
S3:	exploration loop (frontier expansion)
S4:	classification loop (candidate x dimension)
S4A:	adaptation loop (iterated updates) [if active]
S5:	emit finding (+ optional meta)
S6:	emit exactly one error (+ optional meta)

Annex F — Change Control and Deprecation Policy

F.1 Purpose

This annex defines how protocol versions, registry schemas, and identifiers evolve without breaking interoperability.

F.2 Change Categories

F.2.1 Editorial Changes

Examples:

- typo fixes,
- formatting,
- clarifications without semantic effect.

Rule: Editorial changes MUST NOT require protocol version increments.

F.2.2 Backward-Compatible Additions

Examples:

- adding optional fields,
- adding new transforms or dimensions with new identifiers.

Rule: Backward-compatible additions MAY be introduced as:

- minor protocol revision, or
- registry snapshot update, provided semantics of existing identifiers remain unchanged.

F.2.3 Breaking Changes

Examples:

- changing required wire fields,
- changing meaning of existing identifiers,
- altering determinism requirements.

Rule: Breaking changes MUST require a new major protocol version.

F.3 Deprecation Procedure

If an element is deprecated (transform/dimension/context):

- Mark as deprecated in registry snapshot metadata.
- Define a sunset date/version after which it MUST NOT be used.
- Provide migration guidance:

  - replacement identifier(s),
  - expected output differences.

Analyzer behavior

- Before sunset: Analyzer SHOULD support deprecated elements.
- After sunset: Analyzer MUST fail deterministically with:
  - ERROR.phase = VALIDATING
  - ERROR.code = UNKNOWN_REFERENCE or a dedicated deprecation code if specified.

F.4 Conformance Impact

If a deprecated element is used:

- L1 may accept it (if still supported),
- L2+ MUST remain deterministic,
- L3+ MUST preserve adaptive guarantees if adaptation is involved.

F.5 Change Log Requirements

Each release of:

- this Implementation Profile, and
- the wire protocol specification,
  MUST include:
  - a normative change log,
  - a compatibility statement,
  - the effective date of changes.