MODIVX STANDARD DEFINITION BINDING SPECIFICATION

Protocol version	1.0
Document version	2.0
Status	Released
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

Status of This Document (Required Inputs and Comparability)

The Binding Specification standardizes the mandatory accompanying information (required metadata) as well as the parameters used to determine whether two test runs are indeed comparable. Rationale for the requirement: Without these minimum specifications, it is impossible to reliably determine whether results were generated under identical or sufficiently similar boundary conditions.

The MODIVX **Base Standard** defines the diagnostic framework and semantics, while the **DCO Profile** specifies additional deterministic and canonicalization requirements for Interoperability. All requirements with the [PROFILE: DCO] label are excluded from the Base Standard.


RUBEN JAYBIRD
INSTITUTE

1. Scope

1.1 Purpose

This document normatively specifies how a Wire Protocol (WP) Analyzer MUST integrate and invoke a Mathematical Core (MC) Engine in order to produce interoperable, deterministic wire-visible outputs. **[PROFILE: DCO]**

This binding specification makes the MC Engine operationally normative for Analyzer behavior, without modifying the MC documents.

1.2 What this document defines

This document defines, normatively:

1. a binding data model (types, canonical representations, errors),
2. a required MC Engine interface (functions, inputs/outputs, determinism),
3. the binding of Engine calls to the Analyzer execution flow,
4. deterministic selection rules required for adaptation binding, and
5. mapping rules and disclosures required for interoperability.

2. Conformance Language

The key words MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, and MAY are to be interpreted as described in RFC 2119.

3. Normative References

The following documents are normative references:

- [MC-CORE] MLD Mathematical Core
- [MC-SPEC] MLD Implementation Specification
- [MC-REF] MLD Reference Implementation
- [WIRE] MIDOVX Wire Protocol
- [PROFILE] MODIVX Implementation Profile

If this document introduces binding requirements not explicitly present in [WIRE] or [PROFILE], those requirements apply in addition to [WIRE]/[PROFILE] and MUST be followed by conforming implementations.

4. Data Model and Type System

This chapter is the canonical type system for the binding layer. All subsequent chapters MUST use these definitions.

4.1 Primitive Types

4.1.1 Bytes

Bytes is an octet sequence.

4.1.2 UTF-8 String

String is a Unicode string encoded as UTF-8 for wire serialization.

4.1.3 Integer

Int is a signed integer. $UInt$ is an unsigned integer.

4.1.4 Hash [PROFILE: DCO]

Hash is Bytes with a declared algorithm identifier.

```
type Hash = {
  alg: String,    // e.g., "sha256"
  value: Bytes
}
```

Hash.value MUST be the digest of the specified algorithm over the specified input bytes.

4.2 Core Binding Types

4.2.1 EqClass

EqClass is the canonical representation of an MC equivalence class.

```
type EqClass = Bytes
```

Requirements

- EqClass MUST be canonical bytes as defined by [MDL Specification Specs] canonicalization rules.
- Two EqClass values MUST be byte-equal if they represent the same canonical MC equivalence class. **[PROFILE: DCO]**

4.2.2 CanonKey [PROFILE: DCO]

CanonKey is a deterministic ordering key derived from an EqClass.

```
type CanonKey = Bytes
```

Requirements

- CanonKey(eq) MUST be deterministic.
- CanonKey(eq) MUST be stable across platforms and implementations given the same canonicalization regime.

4.2.3 Dimension Identifier [PROFILE: DCO]

A dimension identifier is:

```
type DimId = String
```

Requirements

- DimId MUST be the canonical dimension identifier as resolved from the Registry snapshot per [PROFILE].
- DimId comparison for ordering MUST be based on canonical String bytes.

4.2.4 Invariant Identifier [PROFILE: DCO]

```
type InvId = String
```

Requirements

- Invariant identifiers MUST be stable within the declared registry_snapshot_id.

4.2.5 State Identifier

A diagnostic state identifier is:

```
type StateId = String
```

Requirements

- StateId MUST be one of the declared states for that dimension (registry-bound).
- Ordering of states within a dimension MUST be defined by the resolved dimension definition (posset or declared linear extension).

4.2.6 Theta Identifier (Meta Scheme Identifier)

```
type ThetaId = String
```

4.3 Diagnostic Profile Types

4.3.1 ProfileVector

A diagnostic profile vector is represented as a deterministic dimension-indexed mapping:

```
type ProfileVector = {
  dims: list[DimId],          // deterministic order
  states: list[StateId]       // same length as dims
}
```

Requirements

- dims MUST contain each dimension exactly once.
- dims order MUST equal the binding parameter dim_order (see §5). **[PROFILE: DCO]**
- states[k] MUST be the state for dimension dims[k].

4.4 Context Types

4.4.1 ResolvedContext [PROFILE: DCO]

ResolvedContext is the fully resolved context that the Analyzer passes to the MC Engine. It is intentionally explicit to prevent hidden non-determinism.

```
type ResolvedContext = {
  context_id: String,                       // canonical registry id
  context_fingerprint: Hash,                // per [WIRE]/[PROFILE]
  purpose_params: map[String, String],      // canonicalized, key-sorted
  transform_pipeline_id: String,            // canonical id or "none"
  transform_fingerprint: Hash,              // canonical pipeline hash
  coverage_mode: String,                    // registry/standard enum
  allow_partial: bool,
  depth_limit: UInt,                        // if applicable to exploration
  max_variants: UInt                        // if applicable
}
```

Requirements

- All fields MUST be deterministically derived from the request + registry snapshot. **[PROFILE: DCO]**
- purpose_params keys MUST be canonicalized and sorted lexicographically by UTF-8 bytes. **[PROFILE: DCO]**
- Fingerprints MUST be computed deterministically and MUST be stable across implementations. **[PROFILE: DCO]**

This type may carry more fields than are strictly necessary for MC but MUST include all fields that affect determinism and comparability of results.

4.5 Result and Error Model

4.5.1 Result[T]

All MC Engine calls in this binding layer MUST use:

```
type Result[T] = Ok(T) | Err(EngineError)
```

4.5.2 EngineError

EngineError is an internal error suitable for deterministic mapping to the Analyzer failure/error regime.

```
type EngineError = {
  phase: EnginePhase,
  code: EngineErrorCode,
  message: String,                // stable, non-localized
  details: map[String, String]    // stable key/value, optional
}
```

Requirements

- message MUST be stable and MUST NOT depend on locale.
- details keys MUST be canonicalized and sorted for serialization.
- Errors MUST be deterministic for identical inputs where feasible (i.e., not including random tokens, memory addresses, timestamps).

4.5.3 EnginePhase

Engine phases are an enum aligned to wire/profile semantics:

```
enum EnginePhase {
  VALIDATING,
  EXPLORING,
  EVALUATING,
  ADAPTING,
  GENERAL
}
```

4.5.4 EngineErrorCode

Engine error codes are canonical identifiers (strings):

```
type EngineErrorCode = String
```

Minimum required set

The implementation MUST support at least the following codes:

- MALFORMED_INPUT
- UNKNOWN_DIMENSION

- INVARIANT_EVALUATION_FAILED
- CLASSIFICATION_UNDEFINED
- EXPLORATION_BUDGET_EXCEEDED
- PROGRESS_VIOLATION
- INTERNAL_ENGINE_ERROR
- UPDATE_UNDEFINED
- UPDATE_NOT_REALIZABLE
- ADAPTATION_BUDGET_EXCEEDED

Additional codes MAY be defined but MUST remain stable within a version.

4.6 Deterministic Ordering Conventions [PROFILE: DCO]

4.6.1 Canonical sorting for maps/sets

When this document requires deterministic ordering:

- String keys MUST be sorted lexicographically by UTF-8 bytes.
- EqClass values MUST be ordered by CanonKey(EqClass) ascending (bytewise).

4.6.2 Tie-break semantics (binding-wide)

Whenever this document requires a deterministic selection among multiple admissible candidates, the binding-wide default is:

1. minimal CanonKey(EqClass)
2. if still tied, minimal dimension order index (dim_order)
3. if still tied, stable bytewise minimal serialization of the candidate record

This ordering MUST be treated as deterministic selection only and MUST NOT be interpreted as a semantic ranking of diagnostic profiles.

5. Binding Parameters

This chapter defines the complete binding parameter set (BindParams) used by the Analyzer when invoking the MC Engine. BindParams exists to ensure that all Engine outputs are fully deterministic, wire-comparable, and reproducible.

All subsequent chapters MUST treat BindParams as part of the effective input to MC computation.

5.1 BindParams Object

5.1.1 Type definition

```
type BindParams = {
  registry_snapshot_id: String,          // REQUIRED
  dim_order: list[DimId],                // REQUIRED
  max_depth: UInt,                       // REQUIRED
  max_steps: UInt,                       // REQUIRED
  tie_break_policy: TieBreakPolicy,      // REQUIRED
  approx_policy: ApproxPolicy,           // REQUIRED
  run_mode: RunMode                      // REQUIRED
}
```

5.1.2 Requirements

1. BindParams MUST be passed to every MC Engine call defined in this specification.
2. BindParams MUST be derived deterministically from:
   - the DIAGNOSE_REQUEST,
   - the resolved registry snapshot, and
   - Analyzer configuration parameters that are explicitly declared as part of the binding regime.
3. If two implementations share the same:
   - request bytes (canonicalized per [WIRE]/[PROFILE]),
   - registry_snapshot_id,
   - and the same binding parameter values,
   - then they MUST compute identical binding-layer outcomes, subject to the conformance requirements of [PROFILE] and this binding spec.

5.2 Parameter Sources and Resolution

5.2.1 Source hierarchy

Each BindParams field MUST originate from exactly one of the following sources:

1. Request-supplied (carried in DIAGNOSE_REQUEST),
2. Registry-supplied (from resolved registry snapshot),
3. Analyzer configuration (local, but MUST be declared in an implementation manifest and MUST affect comparability fingerprints where required).

5.2.2 Determinism rule (no hidden inputs)

The Analyzer MUST NOT allow any additional hidden parameter to affect MC Engine outputs. In particular, it MUST NOT use:

- current time,
- random seeds (unless explicitly part of config and disclosed),
- environment variables,

- thread scheduling nondeterminism,
- as effective inputs to the MC Engine.

If a local configuration parameter affects MC Engine behavior, it MUST:

- be part of BindParams or be included in ResolvedContext,
- and be included in comparability/disclosure fields per §10.

5.3 Field Semantics

5.3.1 registry_snapshot_id

Type: String

Requirements

- MUST be the canonical identifier of the immutable snapshot resolved according to [PROFILE].
- MUST be included in all wire-visible outputs that claim determinism/comparability (see §9).

5.3.2 dim_order

Type: list[DimId]

Meaning

A deterministic iteration order over dimensions.

Requirements

1. MUST contain each active diagnostic dimension exactly once.
2. MUST be derived deterministically from the registry snapshot:
   - either as an explicit registry-provided order, or
   - as lexicographic sort of DimId values (UTF-8-byte order),
   - depending on the Analyzer profile configuration.
3. MUST NOT be interpreted as a semantic ranking of dimensions.
4. MUST be used as the canonical serialization order for:
   - ProfileVector.dims,
   - per-dimension findings in DIAGNOSTIC_FINDING.

5.3.3 max_depth

Type: UInt

Meaning

Maximum reachability / exploration depth used by Reach_i_impl and any other bounded closure operations.

Requirements

- MUST satisfy max_depth ≥ 0.
- If max_depth = 0, Reach_i_impl(eq, z, 0, params) MUST return the singleton set {eq} (no expansion).
- MUST be disclosed per §10.

Default

If not explicitly provided by request or registry, the Analyzer MUST use a deterministic configured default:

DEFAULT_MAX_DEPTH, and this value MUST be disclosed.

5.3.4 max_steps

Type: UInt

Meaning

Maximum number of adaptation iterations for a run (trajectory bound).

Requirements

- MUST satisfy max_steps ≥ 1 for any run that can enter ADAPTING/S4A.
- MUST be disclosed per §10.

Default

If not explicitly provided by request or registry, the Analyzer MUST use deterministic configured default:

DEFAULT_MAX_STEPS, disclosed.

5.3.5 tie_break_policy

Type: TieBreakPolicy

Definition

```
type TieBreakPolicy = {
  primary: String,              // MUST be "CANONKEY_EQCLASS"
  secondary: list[String],      // optional
  tertiary: list[String]        // optional
}
```

Allowed values

primary MUST equal: "CANONKEY_EQCLASS"

Allowed secondary and tertiary elements are enumerated:

- "DIM_ORDER_INDEX"

- "CANDIDATE_SERIALIZATION_MIN"

No other values are permitted unless this binding spec is versioned to include them.

Requirements

1. Tie-break ordering MUST be used only when selecting among candidates already deemed admissible (e.g., Pareto-admissible).
2. Tie-break ordering MUST NOT induce any semantic ranking of diagnostic profiles.
3. The Analyzer MUST apply tie-break ordering consistently across:
   - update selection (§8),
   - candidate ordering for stable outputs (§9),
   - and any deterministic iteration order requirements.

Default

```
tie_break_policy = {
  primary: "CANONKEY_EQCLASS",
  secondary: ["DIM_ORDER_INDEX"],
  tertiary: ["CANDIDATE_SERIALIZATION_MIN"]
}
```

5.3.6 approx_policy

Type: ApproxPolicy

Definition

```
type ApproxPolicy = {
  img_impl_relation: ApproxRelation,              // REQUIRED
  reach_impl_relation: ApproxRelation,            // REQUIRED
  reach_depth_semantics: ReachDepthMode   // REQUIRED
}

enum ApproxRelation {
  UNDER_APPROX,             // implementation returns subset
  OVER_APPROX               // implementation returns superset
}

enum ReachDepthMode {
  IMG_CLOSURE_UP_TO_K, // Reach is computed by iterated Img closure up to k
  ENGINE_DEFINED            // engine-defined but MUST be disclosed precisely
}
```

Requirements

1. img_impl_relation MUST be declared.
2. reach_impl_relation MUST be declared.
3. reach_depth_semantics MUST be declared.

4. If reach_depth_semantics = ENGINE_DEFINED, the Analyzer MUST provide a stable textual definition in the disclosure fields (§10) sufficient for independent reimplementation.

Default

If not provided, ApproxPolicy MUST be deterministically set by Analyzer configuration and disclosed.

5.3.7 run_mode

Type: RunMode

Definition

```
enum RunMode {
  CLASSIFY_ONLY,          // B1 only: compute findings, no adaptation
  ADAPT_IF_NEEDED,        // B2: adapt only if classification indicates need
  FORCE_ADAPT             // B2: run adaptation loop regardless
}
```

Requirements

- The chosen run_mode MUST be disclosed per §10.
- If run_mode = CLASSIFY_ONLY, the Analyzer MUST NOT invoke adaptation functions (e.g., A_i) and MUST NOT enter S4A ADAPTING.

Default

If not explicitly specified, run_mode MUST be ADAPT_IF_NEEDED.

5.4 Canonical Serialization of BindParams [PROFILE: DCO]

To ensure byte-identical behavior where required by [PROFILE], BindParams MUST be serializable in a canonical form.

5.4.1 Canonical BindParams encoding (for hashing/disclosure)

The binding layer MUST define a canonical encoding Enc(BindParams):

- JSON-like canonical encoding, UTF-8, no whitespace,
- map keys sorted lexicographically,
- lists encoded in order.

This encoding MUST be stable across implementations.

5.4.2 BindParams fingerprint

The Analyzer MUST compute:

```
bind_params_fingerprint = Hash("sha256", Enc(BindParams))
```

and expose this fingerprint in wire-visible outputs as defined in §9/§10.

5.5 Completeness and Upgrade Rules

5.5.1 Completeness

An implementation MUST treat BindParams as complete. Any behaviorally relevant parameter not included in BindParams (or in ResolvedContext) is a violation of this binding spec.

5.5.2 Upgrade compatibility [PROFILE: DCO]

Future versions of this binding spec MAY extend BindParams. In that case:

- new fields MUST have deterministic defaults,
- defaults MUST be disclosed,
- and canonical encoding MUST remain unambiguous.

6. MC Engine Interface

This chapter defines the complete and implementable MC Engine interface required by this binding specification.

An Analyzer conforming to this document MUST invoke an Engine that implements this interface exactly (or provides an equivalent adapter that preserves semantics and determinism).

6.1 Interface Principles

6.1.1 Deterministic function contract

All Engine functions defined in this chapter MUST be:

- pure with respect to binding inputs, meaning their outputs MUST be a deterministic function of:
  - EqClass eq,
  - ResolvedContext z,
  - BindParams params.

The Engine MAY use internal caching, but caches MUST NOT change observable results.

6.1.2 Canonical input requirements

Before calling any Engine function, the Analyzer MUST ensure:

- eq is canonical EqClass bytes (§4.2.1),
- z is a fully resolved ResolvedContext (§4.4.1),
- params is fully resolved and canonicalizable (BindParams, §5).

If any of these conditions fail, the Analyzer MUST NOT call the Engine and MUST fail the run with a wire-visible ERROR in phase VALIDATING.

6.1.3 Output stability

For identical inputs (eq, z, params) the Engine MUST produce:

- identical return types (Ok vs Err),
- identical return values (byte-identical after canonical serialization where applicable),
- identical errors (EngineError) including stable message content.

6.2 Engine Function Groups

This interface is organized by binding conformance level:

- B1 (Classification Binding): §6.3
- B2 (Adaptation Binding): §6.4
- B3 (Meta Binding): §6.5

An implementation claiming level B2 MUST implement all B1+ B2 functions.

An implementation claiming level B3 MUST implement all B1+ B2+ B3 functions.

6.3 Required Engine Functions — B1 (Classification Binding)

6.3.1 CanonicalizeEqClass [PROFILE: DCO]

Purpose: ensure canonical equivalence class encoding.

Signature

```
CanonicalizeEqClass(eq_raw: Bytes, z: ResolvedContext, params: BindParams)
  -> Result[EqClass]
```

Requirements

1. MUST return canonical EqClass bytes per [MC-SPEC].
2. MUST return Err with:
    - phase = VALIDATING
    - code = MALFORMED_INPUT when canonicalization fails.

Note

The Analyzer MAY canonicalize outside the Engine. If so, this function MAY be implemented as identity.

6.3.2 Inv_i — invariant evaluation (per dimension)

Signature

```
Inv_i(dim: DimId, eq: EqClass, z: ResolvedContext, params: BindParams)
  -> Result[map[InvId, bool]]
```

Requirements

1. MUST succeed only for valid dim in the active dimension set.
2. MUST return Err with:
    - phase = VALIDATING
    - code = UNKNOWN_DIMENSION when dim is unknown.
3. On computational failure MUST return:
    - phase = EVALUATING
    - code = INVARIANT_EVALUATION_FAILED (or a stable subcode)

Canonicalization

- Returned map keys MUST be canonical InvId strings.
- The Analyzer MUST serialize keys in canonical sorted order (§4.6.1).

6.3.3 delta_i — diagnostic classification (per dimension)

Signature

```
delta_i(dim: DimId, eq: EqClass, z: ResolvedContext, params: BindParams)
  -> Result[StateId]
```

Requirements

1. MUST return a StateId that is valid for the given dim under registry_snapshot_id.
2. MUST be consistent with MC diagnostic semantics ([MC-CORE], [MC-SPEC]) for the declared approximation regime.
3. If classification is undefined (no applicable table/rule), MUST return:
    - phase = EVALUATING
    - code = CLASSIFICATION_UNDEFINED.

6.3.4 Profile — full profile vector (all dimensions)

Signature

```
Profile(eq: EqClass, z: ResolvedContext, params: BindParams)
  -> Result[ProfileVector]
```

Requirements

1. MUST return ProfileVector.dims equal to params.dim_order.
2. MUST return exactly one StateId per dimension.
3. MUST be equivalent to invoking delta_i for each dimension under identical inputs.

Error behavior

If any single dimension fails deterministically, the Engine MAY:

- return Err with the first failing dimension error, or
- return a partial internal structure and let the Analyzer enforce allow_partial behavior.

However, the binding layer wire mapping (Chapter 10) is authoritative for partial completion, not the Engine.

6.4 Required Engine Functions — B2 (Adaptation Binding)

6.4.1 GenP_i — perturbation generator per dimension [PROFILE:DCO]

Signature

```
GenP_i(dim: DimId, z: ResolvedContext, params: BindParams)
  -> Result[list[Perturbation]]
```

Where

```
type Perturbation = Bytes | String
```

Requirements

1. MUST generate a deterministic perturbation set for dim.
2. If the MC Engine uses Variant B semantics (compositions), the perturbation representation MUST be stable and canonical.
3. GenP_i output MUST be ordered deterministically:
    - primary ordering by canonical encoding of Perturbation,
    - secondary ordering by params.tie_break_policy when applicable.

Note

This function exists because Wire/Profile-level comparability requires that exploration/adaptation be repeatable and auditable. If an Engine does not expose perturbations, the binding adapter MUST.

6.4.2 Apply_i — apply perturbation (per dimension)

Signature

```
Apply_i(dim: DimId, p: Perturbation, eq: EqClass, z: ResolvedContext, params: BindParams)
  -> Result[EqClass]
```

Requirements

1. MUST return canonical EqClass bytes.
2. MUST be deterministic.
3. If perturbation is not applicable, MUST either:
    - return Ok(eq) (identity), OR
    - return an Err with stable error code. The chosen convention MUST be stable and disclosed in §10.

6.4.3 Img_i_impl — bounded candidate image

Signature

```
Img_i_impl(dim: DimId, eq: EqClass, z: ResolvedContext, params: BindParams)
  -> Result[list[EqClass]]
```

Requirements

1. MUST return a finite list of canonical EqClass.
2. MUST represent the Engine's declared approx_policy.img_impl_relation.
3. MUST be deterministic.
4. Return list MUST be sorted deterministically by:
    - CanonKey(EqClass) ascending,
    - then by stable bytewise EqClass serialization.

6.4.4 Reach_i_impl — bounded reachability closure

Signature

```
Reach_i_impl(dim: DimId, eq: EqClass, z: ResolvedContext, max_depth: UInt, params: BindParams)
  -> Result[list[EqClass]]
```

Minimum required meaning

MUST represent the closure of iterated Img_i_impl up to max_depth, as declared by params.approx_policy.reach_depth_semantics.

Requirements

1. MUST be deterministic.
2. MUST return a list sorted deterministically using the same ordering as Img_i_impl.
3. MUST NOT include duplicates.

6.4.5 ParetoCompareProfiles

Purpose: provide a dimension-aware dominance comparison consistent with MC ordering.

Signature

```
ParetoCompareProfiles(p: ProfileVector, q: ProfileVector, z: ResolvedContext, params: BindParams)
  -> Result[Dominance]
```

Where

```
enum Dominance { P_DOMINATES_Q, Q_DOMINATES_P, INCOMPARABLE, EQUAL }
```

Requirements

- MUST implement componentwise comparison using the registry-resolved ordering for each dimension.
- MUST be deterministic.

6.4.6 SelectCandidate [PROFILE:DCO]

Purpose: deterministic selection among admissible candidates.

Signature

```
SelectCandidate(candidates: list[EqClass], z: ResolvedContext, params: BindParams)
  -> Result[EqClass]
```

Requirements

1. MUST select deterministically using:
    - primary: minimal CanonKey(EqClass)
    - secondary/tertiary per params.tie_break_policy
2. MUST NOT use profile values for selection beyond admissibility (non-aggregation rule).

6.4.7 A_i — adaptation step (per dimension)

Signature

```
A_i(dim: DimId, eq: EqClass, z: ResolvedContext, theta: ThetaId, params: BindParams)
  -> Result[EqClass]
```

Requirements

1. MUST satisfy the adaptation semantics defined in Chapter 8.
2. MUST return canonical EqClass.
3. MUST return Err with:
    - phase = ADAPTING

- code = PROGRESS_VIOLATION when adaptation would violate required progress constraints.

6.5 Required Engine Functions — B3 (Meta Binding)

6.5.1 MetaObserve

Signature

```
MetaObserve(trace: RunTrace, z: ResolvedContext, params: BindParams)
  -> Result[MetaObservation]
```

Where

```
type RunTrace = {
  steps: list[TraceStep]          // ordered
}
type TraceStep = {
  step_index: UInt,
  eq: EqClass,
  profile: ProfileVector,
  state: String                   // e.g., "S3", "S4", "S4A"
}
type MetaObservation = {
  obs_code: String,               // stable identifier, e.g., "H_CYCLE"
  obs_payload: map[String, String]
}
```

Requirements

1. MUST be deterministic.
2. RunTrace.steps MUST be processed in step_index order.
3. MetaObservation.obs_code MUST be stable and from an Engine-declared observation set.
4. obs_payload keys MUST be canonicalized and deterministically ordered for serialization.

6.5.2 NextTheta

Signature

```
NextTheta(theta: ThetaId, h: MetaObservation, z: ResolvedContext, params: BindParams)
  -> Result[ThetaId]
```

Requirements

1. MUST be deterministic.
2. MUST return a stable ThetaId.
3. If no change is required, SHOULD return the input theta.

6.6 Engine Compliance Requirements

6.6.1 Stability across implementations

For binding interoperability, the Engine MUST satisfy:

- given identical (eq, z, params) the outputs MUST be stable across:
  - OS/platform differences,
  - language/runtime differences,
  - independent builds,
  - subject to the canonicalization requirements.

6.6.2 Canonical set/list discipline

All Engine-returned lists in this chapter MUST:

- be duplicate-free,
- be deterministically ordered,
- use canonical underlying representations.

6.7 Engine Interface Completeness

An implementation claiming conformance to this binding spec MUST implement all functions required by its claimed binding level (B1/B2/B3).

If a function is not natively present in the underlying MC implementation, a binding adapter MAY implement it, but the resulting behavior MUST remain consistent with [MC-CORE]/[MC-SPEC] and with this binding spec.

7. Analyzer ↔ Engine Call Binding

This chapter defines exactly how an Analyzer MUST invoke the MC Engine during a run, and how Engine calls bind to the wire-visible Analyzer state machine defined in [WIRE]/[PROFILE].

This chapter is written to be directly implementable: it specifies call order, minimum required behavior, and required invariants.

7.1 Binding Overview

7.1.1 Execution states

The Analyzer MUST implement the execution states defined in [WIRE]/[PROFILE]:

- S0 IDLE
- S1 REQUESTED
- S2 INITIALIZED
- S3 EXPLORING

- S4 CLASSIFYING
- S4A ADAPTING
- S5 COMPLETED
- S6 FAILED

7.1.2 Binding rule

For the purposes of this binding specification:

- S3 EXPLORING binds to MC Engine candidate generation operations.
- S4 CLASSIFYING binds to MC Engine classification operations.
- S4A ADAPTING binds to MC Engine adaptation/update operations.
- If B3 is enabled, meta calls MAY occur after each adaptation step and/or at termination.

7.2 Run Initialization and Canonical Inputs

7.2.1 S1 REQUESTED (Parse and validate request)

On receipt of a DIAGNOSE_REQUEST, the Analyzer MUST:

1. parse the message per [WIRE],
2. validate required fields per [PROFILE],
3. reject malformed requests by transitioning to S6 FAILED and emitting exactly one ERROR message.

No MC Engine calls are permitted in S1.

7.2.2 S2 INITIALIZED (Resolve registry snapshot and context)

In S2, the Analyzer MUST deterministically construct:

- BindParams params (§5),
- ResolvedContext z (§4.4.1),
- the canonical initial equivalence class EqClass eq0.

Required resolution steps

1. resolve registry_snapshot_id,
2. resolve active dimension set and build params.dim_order,
3. compute context_fingerprint and transform_fingerprint,
4. canonicalize the carrier into eq0:
5. either by calling Engine.CanonicalizeEqClass(eq_raw, z, params) or by applying the identical canonicalization rules externally.

If any step fails, the Analyzer MUST transition to S6 FAILED and emit one ERROR.

7.3 Candidate Generation Binding (S3 EXPLORING)

7.3.1 Purpose

The goal of S3 is to establish a deterministic candidate set C of EqClass objects for possible evaluation and/or adaptation.

7.3.2 Candidate set model

The Analyzer MUST represent the exploration candidate set as:

```
type CandidateSet = {
  root: EqClass,                  // eq0
  candidates: list[EqClass]       // canonical, duplicate-free, ordered
}
```

Requirements

- root MUST equal the canonical initial class eq0.
- candidates MUST include eq0.
- candidates MUST be duplicate-free.
- candidates MUST be deterministically ordered by CanonKey(EqClass) ascending.

7.3.3 Exploration strategies

The Analyzer MUST implement at least one of the following exploration strategies:

- STRATEGY A (Image-based): use Img_i_impl per dimension,
- STRATEGY B (Reach-based): use Reach_i_impl per dimension.

The chosen strategy MUST be disclosed per §10.

7.3.4 Mandatory minimum exploration (B1+)

Even when operating in B1 (classification only), the Analyzer MUST ensure:

```
CandidateSet.candidates = [eq0]
```

That is, a trivial candidate set is acceptable for B1.

7.3.5 Exploration for adaptation (B2/B3)

If params.run_mode != CLASSIFY_ONLY, the Analyzer MUST perform bounded exploration:

Inputs

- max_depth = params.max_depth
- active dimensions in params.dim_order

Required algorithm (normative)

The Analyzer MUST compute:

1. Initialize: C = {eq0}
2. For each dim in params.dim_order in order:

- If strategy is STRATEGY A:
  - call Engine.Img_i_impl(dim, eq0, z, params) to obtain I_dim
  - add all members of I_dim to C

- If strategy is STRATEGY B:
  - call Engine.Reach_i_impl(dim, eq0, z, params.max_depth, params) to obtain R_dim
  - add all members of R_dim to C

3. Canonicalize:
   - remove duplicates (byte equality),
   - sort C by CanonKey(EqClass) ascending,
   - output as CandidateSet.candidates.

Constraints

The Analyzer MUST NOT exceed the exploration budgets declared by:

- params.max_depth,
- and any additional bounds enforced by [PROFILE] (e.g., depth_limit/max_variants in ResolvedContext).

If exploration terminates due to bounds, the Analyzer MUST record termination metadata for disclosure (§10).

Failure behavior

- Any Engine exploration error MUST be mapped to ERROR.phase = EXPLORING unless the error is explicitly VALIDATING or GENERAL.
- If allow_partial=false, exploration failure MUST result in S6 FAILED.
- If allow_partial=true, the Analyzer MAY continue with a reduced candidate set (at minimum {eq0}), but MUST disclose the truncation reason (§10) and MUST mark findings as partial (§9).

7.4 Evaluation Binding (S4 CLASSIFYING)

7.4.1 Purpose

In S4, the Analyzer MUST compute diagnostic states for at least one EqClass, and prepare the wire-visible DIAGNOSTIC_FINDING.

7.4.2 Evaluation target selection

The Analyzer MUST choose the evaluation target eq_eval as follows:

- If params.run_mode = CLASSIFY_ONLY: eq_eval = eq0.
- Otherwise: eq_eval MUST be the current adaptation state eq_current (initially eq0, then updated in S4A).

7.4.3 Required evaluation call

The Analyzer MUST call:

```
Engine.Profile(eq_eval, z, params)
```

and obtain:

- Ok(ProfileVector p_eval), or
- Err(EngineError e).

7.4.4 Evaluation error handling

If Engine.Profile returns Err(e):

- If allow_partial=false: MUST transition to S6 FAILED and emit ERROR.
- If allow_partial=true: MUST produce a partial finding as defined in §9, including:
  - which dimensions failed (if deterministically available),
  - and the error code(s) contributing to partial completion.

Note: The Engine MAY expose per-dimension failure details by additional diagnostic APIs; if so, those details MUST be used deterministically.

7.5 Adaptation Binding (S4A ADAPTING) — B2/B3

7.5.1 Preconditions

The Analyzer MUST enter S4A only if:

- params.run_mode != CLASSIFY_ONLY, and
- adaptation is required or forced by params.run_mode.

7.5.2 Adaptation loop model

The Analyzer MUST maintain:

```
type AdaptationState = {
  eq_current: EqClass,
  step_index: UInt,       // 0-based
  theta: ThetaId,
  trace: RunTrace         // B3 only
}
```

Initialization:

- eq_current = eq0
- step_index = 0
- theta = initial theta (registry-defined or engine-default; MUST be disclosed)
- trace.steps = [] if B3 enabled

7.5.3 Required per-step behavior

For each adaptation step while step_index < params.max_steps:

1. Evaluate current state (S4):
   call Engine.Profile(eq_current, z, params) → p_current
2. Generate candidate updates
   for each dim in params.dim_order:
   - call Engine.A_i(dim, eq_current, z, theta, params) → eq_dim
   - collect all eq_dim into candidate list U
3. Admissibility filtering
   - compute Profile(eq) for each candidate eq in U
   - keep only those candidates that satisfy the admissibility predicate:
     - p_current is not worsened componentwise and at least one dimension improves
     - exact semantics defined in Chapter 8 (Normative Adaptation Semantics)
4. Selection
   - if admissible set is empty: terminate adaptation (fixpoint)

     else call:

     Engine.SelectCandidate(admissible_eqs, z, params) → eq_next
5. Update
   - set eq_current = eq_next
   - increment step_index

Budget constraints

The Analyzer MUST NOT perform more than:

- params.max_steps update steps,
- and MUST honor any additional [PROFILE] time/budget ceilings.

If budget ceiling triggers termination, this MUST be disclosed (§10).

7.5.4 Cycle detection (required)

The Analyzer MUST implement deterministic cycle detection over EqClass bytes:

- Maintain a set VisitedEq = {eq0}.
- Before accepting eq_next, if eq_next ∈ VisitedEq, a cycle is detected.

On cycle

- terminate adaptation (cycle termination),
- record termination_reason = "CYCLE_DETECTED" for disclosure (§10),
- proceed to final evaluation and output (S4 → S5).

Cycle detection MUST be deterministic and MUST not depend on concurrency.

7.5.5 Fixpoint detection (required)

A fixpoint is detected when:

- admissible candidate set is empty, OR
- eq_next == eq_current.

On fixpoint:

- terminate adaptation,
- set termination_reason = "FIXPOINT",
- proceed to output.

7.5.6 Adaptation error handling

If any A_i call returns Err(e):

If allow_partial=false: MUST transition to S6 FAILED and emit ERROR with phase ADAPTING.

If allow_partial=true: Analyzer MAY:

- drop that candidate update, continue with remaining candidates, and produce partial completion,
- OR terminate adaptation early and produce partial completion.

In either case the Analyzer MUST disclose the error contribution in §10 and MUST mark finding partial (§9).

7.6 Meta Binding (B3)

7.6.1 Trace construction (required for B3)

If binding level B3 is claimed, the Analyzer MUST populate RunTrace deterministically.

At minimum, at each completed evaluation step, append:

```
TraceStep = {
  step_index: step_index,
```

```
  eq: eq_current,
  profile: p_current,
  state: "S4"
}
```

7.6.2 Meta observation cadence

The Analyzer MUST perform meta observation either:

- after each adaptation step, OR
- at termination,

but MUST choose exactly one cadence deterministically per run and disclose it (§10).

7.6.3 Meta observation and theta update

When meta observation is performed:

1. call Engine.MetaObserve(trace, z, params) → h
2. call Engine.NextTheta(theta, h, z, params) → theta_next
3. update theta = theta_next

Any meta update MUST be disclosed in the meta finding mapping (§9).

7.7 Completion Binding (S5 COMPLETED / S6 FAILED)

7.7.1 Normal completion (S5)

The Analyzer MUST transition to S5 COMPLETED when:

- evaluation succeeded (or partial completion permitted and produced), and
- no fatal error occurred.

In S5, the Analyzer MUST emit:

- exactly one DIAGNOSTIC_FINDING, and
- optionally one META_DIAGNOSTIC_FINDING if B3 is enabled and meta data exists.

7.7.2 Failure (S6)

The Analyzer MUST transition to S6 FAILED when:

- allow_partial=false and any required phase fails, OR
- request/registry resolution fails, OR
- determinism/canonicalization constraints cannot be satisfied.

In S6, the Analyzer MUST emit exactly one ERROR message.

7.8 Deterministic Call Ordering Guarantees [PROFILE:DCO]

To preserve reproducibility, the Analyzer MUST adhere to:

- dimension iteration in params.dim_order,
- candidate ordering by CanonKey(EqClass),
- selection via Engine.SelectCandidate only.

The Analyzer MUST NOT:

- parallelize Engine calls in a way that changes effective ordering,
- or make nondeterministic early-exit choices based on runtime timing.

Parallel execution MAY be used only if results are merged deterministically.

8. Adaptation Semantics

This chapter defines the binding-level, normative semantics of adaptation/update behavior. It is the normative bridge from MC adaptation concepts into Analyzer execution.

This chapter applies to binding conformance levels B2 and B3.

8.1 Core Requirements

8.1.1 Non-aggregation rule (mandatory)

The Analyzer MUST NOT introduce a semantic total order over full profile vectors.

Concretely:

- The Analyzer MUST use componentwise admissibility (Pareto semantics) to filter candidates.
- The Analyzer MUST use tie-break ordering only as a deterministic selector among already admissible candidates.

This rule applies even if an implementation internally computes scalar scores (it MUST NOT use them for semantic selection unless explicitly permitted by a future version of this binding spec).

8.2 Definitions

8.2.1 Profile dominance

Given two profile vectors:

- p = Profile(eq_p)
- q = Profile(eq_q)

Define dominance using the per-dimension state order relation defined by the registry snapshot.

Dominance is determined by calling:

```
Engine.ParetoCompareProfiles(p, q, z, params) -> Dominance
```

which MUST behave as:

- P_DOMINATES_Q if p is not worse than q in every dimension and strictly better in at least one
- Q_DOMINATES_P analogously
- EQUAL if identical in all dimensions
- INCOMPARABLE otherwise

8.2.2 Improvement and non-regression

Let p_current be the profile of the current state.

A candidate profile p_cand is:

non-regressing if:

```
Engine.ParetoCompareProfiles(p_current, p_cand) != Q_DOMINATES_P
```

i.e., the candidate does not strictly worsen the current profile.

improving if:

```
Engine.ParetoCompareProfiles(p_current, p_cand) == P_DOMINATES_Q
```

where p_cand dominates p_current.

8.2.3 Admissible update

A candidate update eq_cand is admissible iff:

1. it is non-regressing relative to eq_current, and
2. it is improving relative to eq_current.

Equivalently, admissibility requires:

- candidate MUST NOT be worse in any dimension, and
- candidate MUST be strictly better in at least one dimension.

8.3 Candidate Update Construction

8.3.1 Update proposals

At adaptation step t, for each dimension dim the Analyzer MUST compute:

```
eq_dim = Engine.A_i(dim, eq_current, z, theta, params)
```

and collect these into the update proposal set U.

U MUST be treated as a set (duplicates removed).

8.3.2 Candidate pool (normative definition)

The update candidate pool is:

U = { eq_dim | dim ∈ params.dim_order and A_i returned Ok(eq_dim) }

If all A_i calls fail and allow_partial=false, the run MUST fail (S6).

If allow_partial=true, the Analyzer MAY terminate adaptation and produce a partial finding.

8.4 Admissibility Filtering

8.4.1 Required evaluation

For each eq_cand ∈ U, the Analyzer MUST compute:

p_cand = Engine.Profile(eq_cand, z, params)

If Engine.Profile fails for a candidate:

- the candidate MUST be discarded from admissibility consideration, and
- the failure MUST be disclosed if it affects partial completion (§10).

8.4.2 Required admissibility predicate

The admissible set A MUST be computed as:

A = { eq_cand ∈ U | Profile(eq_cand) strictly improves Profile(eq_current) }

i.e., all candidates that satisfy §8.2.3.

8.5 Multi-candidate Semantics (Pareto Set)

8.5.1 Pareto maximality within admissible updates

Even after admissibility filtering, A may contain multiple updates.

The Analyzer MUST compute the Pareto maximal subset:

M = ParetoMax(A)

where eq_x ∈ M iff there does not exist eq_y ∈ A such that:

- Profile(eq_y) dominates Profile(eq_x) strictly.

8.5.2 Normative algorithm for ParetoMax

The Analyzer MUST implement the following deterministic procedure:

```
function ParetoMax(A):
  M = []
  for eq_x in sort_by_canonsel(A):
    dominated = false
    for eq_y in A:
      if Engine.ParetoCompareProfiles(Profile(eq_y), Profile(eq_x)) == P_DOMINATES_Q:
        dominated = true
        break
    if not dominated:
      M.append(eq_x)
  return sort_by_canonsel(M)
```

Where sort_by_canonsel orders EqClass by:

- CanonKey(EqClass) ascending
- tie-break policy secondary keys if applicable

Implementations MAY use a more efficient algorithm as long as the resulting set equals the definition and ordering is deterministic.

8.6 Deterministic Selection [PROFILE:DCO]

8.6.1 Selection scope

The Analyzer MUST select the next state only from the Pareto maximal set M.

No candidate outside M is permitted.

8.6.2 Required selection rule

The Analyzer MUST select:

```
eq_next = Engine.SelectCandidate(M, z, params)
```

and the Engine MUST behave consistently with §6.4.6:

- primary: minimal CanonKey(EqClass)
- secondary/tertiary per params.tie_break_policy

8.6.3 Selection does not create semantic ordering

The SelectCandidate ordering MUST NOT be interpreted as a semantic ranking of profiles.

It is strictly a deterministic selection function.

Wire-level artifacts that expose selection indices (e.g., rank_in, rank_out) MUST treat them as non-semantic determinism aids and MUST NOT interpret them as profile rankings.

8.7 Progress Conditions and Violations

8.7.1 Mandatory progress constraint

If M is non-empty, eq_next MUST satisfy admissibility relative to eq_current.

If selection yields a non-admissible candidate (should be impossible if rules are applied), the Analyzer MUST:

- treat this as a binding violation,
- transition to S6 FAILED,
- emit ERROR.phase = ADAPTING,
- ERROR.code = PROGRESS_VIOLATION.

8.7.2 No-op updates

If eq_next == eq_current, the Analyzer MUST treat this as fixpoint termination (see §7.5.5), not as progress.

8.8 Termination Reasons

The adaptation loop MUST terminate with exactly one reason:

- "FIXPOINT" — no admissible updates exist
- "CYCLE_DETECTED" — an EqClass repeats
- "MAX_STEPS_REACHED" — params.max_steps reached
- "BUDGET_EXCEEDED" — other profile-enforced bound triggered
- "ERROR" — fatal error (S6)

The Analyzer MUST disclose the termination reason in findings (§10).

8.9 Formal Equivalence Note (Non-normative)

The above adaptation semantics correspond to the usual definition:

- admissible updates are strict Pareto improvements relative to current profile
- selection is deterministic tie-break over the Pareto maximal admissible set

Mathematically, if p and q are profile vectors:

- admissible if $\mathrm{p} \prec \mathrm{q}$ componentwise
- ParetoMax retains undominated elements
- selection is $\arg\min \mathrm{CanonKey}$

This note is provided for consistency checking only.

9. Wire Mapping

This chapter specifies the complete, field-by-field mapping from binding-layer objects (MC Engine outputs + run metadata) into the wire-visible messages defined in [WIRE].

This chapter is normative and fully deterministic: given identical (eq, z, params) and identical run trace/termination metadata, two conforming implementations MUST emit equivalent (and where required by [PROFILE], byte-identical after canonicalization) messages.

9.1 Message Emission Rules (Global)

9.1.1 Emission cardinality

Per request, the Analyzer MUST emit exactly one of the following terminal outputs:

- Success path: one DIAGNOSTIC_FINDING (REQUIRED) and optionally one META_DIAGNOSTIC_FINDING (B3 only), then stop, OR
- Failure path: one ERROR, then stop.

The Analyzer MUST NOT emit both DIAGNOSTIC_FINDING and ERROR for the same run.

9.1.2 Deterministic serialization

All emitted messages MUST be serialized using:

- the canonicalization and ordering rules of [PROFILE], and
- the deterministic ordering conventions in §4.6.

9.1.3 Required disclosure attachment

All messages that include findings MUST include the binding fingerprints required by §9.6 and disclosures required by Chapter 10.

9.2 DIAGNOSTIC_FINDING Mapping

9.2.1 Required inputs

To construct a DIAGNOSTIC_FINDING, the Analyzer MUST have:

- eq_final : EqClass — the evaluated equivalence class at termination
- p_final : ProfileVector — the evaluated profile vector (if available)
- z : ResolvedContext
- params : BindParams
- run metadata (termination_reason, budgets, partial flags)

9.2.2 DiagnosticFinding wire object (binding view)

This binding spec assumes the following conceptual wire object fields (names aligned to [WIRE]):

```
DIAGNOSTIC_FINDING = {
  request_id: String,
  registry_snapshot_id: String,
  context_id: String,
  context_fingerprint: Hash,
  bind_params_fingerprint: Hash,

  carrier_equivalence_id: CarrierEquivalenceId,
type CarrierEquivalenceId = {
  equiv_scheme_id: String,
  class_representative_id: String
}
  canonical_carrier_hash: Hash,

  termination: TerminationInfo,
  partial: PartialInfo,

  dimension_findings: list[DimensionFinding]
}
```

If the actual [WIRE] schema uses different field names, implementations MUST map these fields to their [WIRE]-defined equivalents without semantic loss.

9.2.3 Header and binding identity fields

Field	Value source	Requirements
request_id	from DIAGNOSE_REQUEST	MUST be identical to request
registry_snapshot_id	params.registry_snapshot_id	REQUIRED
context_id	z.context_id	REQUIRED
context_fingerprint	z.context_fingerprint	REQUIRED
bind_params_fingerprint	Hash(Enc(BindParams)) (§5.4)	REQUIRED

9.2.4 Carrier / EqClass fields

Field	Value source	Requirements
carrier_equivalence_id	eq_final	MUST equal canonical EqClass bytes
canonical_carrier_hash	Hash("sha256", CanonKey(eq_final))	REQUIRED; algorithm MUST match declaration

Wire requires carrier_equivalence_id as the pair (equiv_scheme_id, class_representative_id).

The Analyzer MUST set:

- equiv_scheme_id = z.context_id (or the registry-resolved equivalence scheme id, if distinct),
- class_representative_id = Base64Url (CanonKey(eq_final)).

Notes

- carrier_equivalence_id is the primary identity of the evaluated carrier equivalence class.
- canonical_carrier_hash exists for stable indexing and comparability.

9.3 Per-Dimension Mapping

9.3.1 DimensionFinding wire object

```
DimensionFinding = {
  dim_id: DimId,
  state: StateId,
  valid: bool,
  invariants: map[InvId, bool],     // optional but deterministic if present
  justification: map[String, String]// optional stable metadata
}
```

9.3.2 Required ordering

dimension_findings MUST be ordered exactly as params.dim_order.

9.3.3 Required fields: dim_id, state, valid

For each dimension dim = params.dim_order[k]:

- dim_id = dim
- If evaluation succeeded for this dimension:
- state = p_final.states[k]
- valid = true

If evaluation failed for this dimension:

- state MUST be set to a registry-defined sentinel state OR omitted if [WIRE] permits omission.
- valid = false
- failure MUST be reflected in partial/termination fields (§9.5)

9.3.4 Optional invariants

If the Analyzer emits invariants:

- MUST call Engine.Inv_i(dim, eq_final, z, params)

- MUST serialize invariant map in canonical key order
- MUST ensure determinism and stability

If invariants are omitted, they MUST be omitted uniformly for the declared conformance profile (no per-run randomness).

9.4 Termination Mapping

9.4.1 TerminationInfo object

```
TerminationInfo = {
  completed_state: String,       // "S5"
  termination_reason: String,    // per §8.8
  step_count: UInt,
  max_steps: UInt,
  depth_reached: UInt,
  max_depth: UInt,
  budgets: map[String, UInt]     // optional: explored_nodes, explored_variants, ...
}
```

9.4.2 Field mapping rules

Field	Value source	Requirements
completed_state	literal "S5"	REQUIRED
termination_reason	from adaptation loop (§8.8) or "CLASSIFY_ONLY"	REQUIRED
step_count	number of executed adaptation steps	REQUIRED
max_steps	params.max_steps	REQUIRED
depth_reached	max depth actually explored	REQUIRED
max_depth	params.max_depth	REQUIRED

If params.run_mode = CLASSIFY_ONLY, then:

- termination_reason = "CLASSIFY_ONLY"
- step_count = 0
- depth_reached = 0

9.5 Partial Results Mapping (allow_partial)

9.5.1 PartialInfo object

```
PartialInfo = {
  allow_partial: bool,
  is_partial: bool,
  reason_code: String,                  // stable identifier
  missing_dimension_ids: list[DimId],   // deterministic order
  contributing_errors: list[ErrorStub]
}

ErrorStub = {
  phase: EnginePhase,
```

```
  code: EngineErrorCode
}
```

9.5.2 Rules

1. partial.allow_partial MUST equal z.allow_partial.
2. If z.allow_partial = false, then:
   - partial.is_partial MUST be false
   - and all dimension findings MUST have valid=true (else run fails and emits ERROR).
3. If any dimension has valid=false, then:
   - partial.is_partial MUST be true
   - missing_dimension_ids MUST list all failed dimensions in params.dim_order order.
4. reason_code MUST be stable and MUST be one of:
   - "EXPLORATION_TRUNCATED"
   - "EVALUATION_FAILED"
   - "ADAPTATION_TRUNCATED"
   - "BUDGET_EXCEEDED"
   - "ENGINE_ERROR"
   - "OTHER"
5. contributing_errors MUST list the stable (phase, code) pairs that caused partiality, deterministically ordered by:
   - phase enum order, then code string order.

9.6 Binding Fingerprints and Comparability [PROFILE:DCO]

9.6.1 Required fingerprints

Every DIAGNOSTIC_FINDING MUST include:

- context_fingerprint (from ResolvedContext)
- bind_params_fingerprint (from BindParams)

These fingerprints define the comparability scope. Two findings MUST NOT be claimed comparable unless these fingerprints match bytewise.

9.6.2 Equivalence vs identity

- Wire equivalence MUST follow [WIRE] equivalence rules (ordinal equivalence).
- Byte identity (L2+) MUST follow [PROFILE] canonicalization rules.

This binding spec requires:

if [PROFILE] level mandates byte-identical behavior, then all fields above MUST be serialized canonically and deterministically.

9.7 META_DIAGNOSTIC_FINDING Mapping (B3)

9.7.1 When required

If binding level B3 is claimed, the Analyzer MUST emit exactly one META_DIAGNOSTIC_FINDING if:

- any meta observation was performed, OR
- theta was changed during the run, OR
- meta disclosure is required by implementation profile.

9.7.2 MetaDiagnosticFinding wire object

```
META_DIAGNOSTIC_FINDING = {
  request_id: String,
  registry_snapshot_id: String,
  context_fingerprint: Hash,
  bind_params_fingerprint: Hash,

  meta_trajectory: list[MetaStep]
}

MetaStep = {
  step_index: UInt,
  theta_before: ThetaId,
  obs_code: String,
  theta_after: ThetaId
}
```

9.7.3 Mapping rules

- meta_trajectory MUST be ordered by step_index ascending.
- At minimum, include one MetaStep at termination.
- If meta updates occurred during adaptation, include each.

Meta fields MUST be stable and deterministic.

9.8 ERROR Mapping (Wire Message)

9.8.1 Emission rule

If the run transitions to S6 FAILED, the Analyzer MUST emit exactly one ERROR message and MUST NOT emit findings.

9.8.2 Error object (binding view)

```
ERROR = {
  request_id: String,
  registry_snapshot_id: String,
  context_fingerprint: Hash,        // if available
  phase: EnginePhase,   // VALIDATING/EXPLORING/EVALUATING/ADAPTING/GENERAL
  code: String,                     // wire registry code id or token
  message: String,
  details: map[String, String]
}
```

9.8.3 Mapping from EngineError

If failure originates from an EngineError e:

- ERROR.phase = e.phase
- ERROR.code = MapEngineErrorCode(e.code) (see Chapter 11)
- ERROR.message = e.message
- ERROR.details = e.details

If failure originates from Analyzer infrastructure (wire parsing, registry resolution), phase MUST be VALIDATING or GENERAL as appropriate.

9.9 Output Determinism Checklist

A conforming Analyzer MUST ensure the following are deterministic for identical inputs:

1. inclusion and ordering of dimension_findings,
2. partial fields including missing dimension lists,
3. termination reason selection and disclosure,
4. fingerprints (context_fingerprint, bind_params_fingerprint, canonical_carrier_hash),
5. meta trajectory ordering and contents (B3),
6. error phase/code mapping.

10. Disclosure and Approximation Reporting

This chapter specifies what an Analyzer MUST disclose in wire-visible outputs to ensure that diagnostic results are interpretable, comparable, and reproducible, even when bounded/approximate operators are used.

This chapter applies to all conformance levels B1–B3.

10.1 Disclosure Principles

10.1.1 No hidden variability

If any parameter, budget, approximation choice, or pruning rule can change:

- candidate sets,
- classifications,
- adaptation outcomes,
- or termination behavior,

then it MUST be disclosed either:

- directly in the output finding(s), OR
- indirectly via a fingerprint that deterministically commits to those parameters.

10.1.2 Required disclosure locations

Disclosure MUST appear in one or more of:

- DIAGNOSTIC_FINDING.termination.budgets,
- DIAGNOSTIC_FINDING.partial,
- an additional disclosure object (if supported by [WIRE]),
- META_DIAGNOSTIC_FINDING (for meta-related disclosures).

If [WIRE] does not define an explicit disclosure field, the Analyzer MUST include disclosures as key/value pairs inside:

- DimensionFinding.justification, OR
- DIAGNOSTIC_FINDING.termination.budgets, using stable keys defined in §10.4.

10.2 Required Binding Disclosures

10.2.1 BindParams disclosure

The Analyzer MUST disclose:

- bind_params_fingerprint (required by §9.6), and
- sufficient information to interpret that fingerprint.

At minimum, the Analyzer MUST disclose the following BindParams fields (in stable key/value form):

- registry_snapshot_id
- max_depth
- max_steps
- run_mode

- tie_break_policy.primary
- tie_break_policy.secondary
- approx_policy.img_impl_relation
- approx_policy.reach_impl_relation
- approx_policy.reach_depth_semantics

If a field is not applicable, it MUST be disclosed as "n/a".

10.2.2 Context disclosure

The Analyzer MUST disclose:

- context_fingerprint
- transform_fingerprint
- coverage_mode
- allow_partial
- depth_limit (if applicable)
- max_variants (if applicable)

If a wire format exposes a transform_manifest_id, it MUST refer to the same transform specification committed to by the disclosed transform_fingerprint.

10.3 Approximation Disclosure (MC Operators)

10.3.1 Img and Reach approximation

If the Engine uses any bounded or approximate operator for:

- Img_i_impl
- Reach_i_impl

then the Analyzer MUST disclose:

1. whether each is an UNDER_APPROX or OVER_APPROX (from ApproxPolicy), and
2. the precise meaning of the depth bound max_depth.

10.3.2 Required statement for Reach semantics

The Analyzer MUST disclose exactly one of:

reach_semantics = "IMG_CLOSURE_UP_TO_K", meaning:

Reach is computed as repeated application of Img_i_impl up to k = max_depth.

or

reach_semantics = "ENGINE_DEFINED", in which case the Analyzer MUST also disclose:

a stable textual definition of reach depth semantics (reach_semantics_definition), sufficient for independent reimplementation.

10.3.3 Candidate truncation disclosure

If exploration is truncated due to budgets, bounds, or policy pruning, the Analyzer MUST disclose:

- exploration_truncated = true
- exploration_truncation_reason (stable code, see §10.5)
- actual depth_reached
- any applicable counters:
- explored_nodes
- explored_variants
- img_calls_total
- reach_calls_total
- If not truncated:
- exploration_truncated = false

10.4 Disclosure Key Registry

All disclosure key names MUST be stable and canonical. The Analyzer MUST use the following keys verbatim (case-sensitive):

10.4.1 Binding keys

- binding.registry_snapshot_id
- binding.bind_params_fingerprint
- binding.max_depth
- binding.max_steps
- binding.run_mode
- binding.tie_break.primary
- binding.tie_break.secondary
- binding.tie_break.tertiary
- binding.approx.img_relation
- binding.approx.reach_relation
- binding.approx.reach_semantics

10.4.2 Context keys

- context.context_fingerprint
- context.transform_fingerprint
- context.coverage_mode

- context.allow_partial
- context.depth_limit
- context.max_variants

10.4.3 Exploration keys

- exploration.strategy // “IMG_BASED” or “REACH_BASED”
- exploration.truncated // “true”/“false”
- exploration.truncation_reason // stable code
- exploration.depth_reached
- exploration.explored_nodes
- exploration.explored_variants

10.4.4 Adaptation keys

- adaptation.termination_reason
- adaptation.step_count
- adaptation.cycle_detected // “true”/“false”
- adaptation.fixpoint_detected // “true”/“false”
- adaptation.max_steps_reached // “true”/“false”

10.4.5 Perturbation keys (Variant B)

- perturbation.mode // “VARIANT_B”
- perturbation.comp_depth
- perturbation.apply_order // “LEFT_TO_RIGHT”
- perturbation.identity_on_inapplicable // “true”/“false”
- 10.4.6 Meta keys (B3)
- meta.enabled // “true”/“false”
- meta.cadence // “EACH_STEP” or “AT_TERMINATION”
- meta.theta_initial
- meta.theta_final
- meta.obs_codes // comma-separated stable list

10.5 Stable Reason Code Registry (Normative)

10.5.1 Exploration truncation reasons

If exploration truncates, exploration.truncation_reason MUST be one of:

- DEPTH_LIMIT_REACHED
- VARIANT_LIMIT_REACHED
- BUDGET_EXCEEDED
- ENGINE_ERROR

- PRUNING_POLICY
- OTHER

10.5.2 Adaptation termination reasons

The binding-level adaptation termination reason MUST be disclosed using:

- FIXPOINT
- CYCLE_DETECTED
- MAX_STEPS_REACHED
- BUDGET_EXCEEDED
- CLASSIFY_ONLY
- ERROR

These codes MUST match the TerminationInfo.termination_reason in §9.4.

10.5.3 Partial completion reasons

PartialInfo.reason_code MUST be one of:

- EXPLORATION_TRUNCATED
- EVALUATION_FAILED
- ADAPTATION_TRUNCATED
- BUDGET_EXCEEDED
- ENGINE_ERROR
- OTHER

10.6 Mandatory Disclosure Sets by Conformance Level

10.6.1 B1

At minimum, B1 implementations MUST disclose:

- all binding keys (§10.4.1),
- all context keys (§10.4.2),
- adaptation termination reason = CLASSIFY_ONLY,
- partial info fields (§9.5), even if not partial.

10.6.2 B2

In addition to B1, B2 MUST disclose:

- exploration strategy and truncation keys (§10.4.3),
- adaptation keys (§10.4.4),
- perturbation keys (§10.4.5).

10.6.3 B3

In addition to B2, B3 MUST disclose:

- meta keys (§10.4.6),
- meta observation codes present in the run.

10.7 Placement Rules (Where to put disclosures)

10.7.1 Preferred placement

If [WIRE] supports an explicit disclosure field, the Analyzer MUST place the key/value disclosure map there.

10.7.2 Fallback placement

If no explicit disclosure field exists, the Analyzer MUST place disclosures in:

1. DIAGNOSTIC_FINDING.termination.budgets (for numeric counters), AND
2. DimensionFinding.justification of a designated dimension "GLOBAL" if supported, otherwise the first dimension in dim_order.

10.7.3 Deterministic ordering

Disclosure maps MUST be serialized in canonical key order (§4.6.1).

10.8 Consistency Requirements

10.8.1 Fingerprint consistency

The disclosed binding.* and context.* fields MUST match the fingerprints included in the finding.

10.8.2 Budget consistency

If termination_reason = BUDGET_EXCEEDED, then:

exploration.truncated MUST be "true" OR adaptation.max_steps_reached MUST be "true".

10.8.3 Perturbation consistency

If perturbation.mode = "VARIANT_B", then:

perturbation.comp_depth MUST be present and numeric.

11. Error Code Binding and Mapping

This chapter defines the complete, deterministic mapping from:

- Engine errors (EngineError, §4.5)
  to
- Analyzer failure identifiers (canonical failure codes)
  to
- Wire-visible ERROR codes (Wire registry identifiers per [WIRE]).

This chapter is normative and applies to all binding levels B1–B3.

11.1 Goals

A conforming Analyzer MUST ensure:

- any fatal error results in exactly one wire ERROR message (§9.8),
- the ERROR.phase is stable and aligned with [WIRE],
- the ERROR.code is stable and maps to the Wire registry where possible,
- mapping behavior is deterministic and reproducible across implementations.

11.2 Canonical Failure Identifier (Binding Layer)

11.2.1 FailureId

The binding layer defines a canonical intermediate identifier:

```
type FailureId = String
```

A FailureId MUST:

- be stable within this binding spec version,
- be suitable for reporting and audit,
- map deterministically to wire ERROR.code.

11.2.2 FailureId formation rule

When an error originates from EngineError e, the Analyzer MUST compute:

```
FailureId = "ENG." + e.phase + "." + e.code
```

Example:

ENG.EVALUATING.CLASSIFICATION_UNDEFINED

When an error originates outside the Engine (e.g., parsing), the Analyzer MUST use:

```
FailureId = "ANZ." + phase + "." + code
```

Example:

ANZ.VALIDATING.MALFORMED_REQUEST

11.3 EngineError → FailureId Mapping Rules

11.3.1 Mapping invariants

For EngineError e, the Analyzer MUST set:

- ERROR.phase = e.phase (exact)
- ERROR.message = e.message
- ERROR.details = e.details

And MUST compute canonical FailureId per §11.2.

11.3.2 Phase validation

If e.phase is not one of:

VALIDATING / EXPLORING / EVALUATING / ADAPTING / GENERAL

then the Analyzer MUST override:

ERROR.phase = GENERAL

and set FailureId = "ENG.GENERAL.INTERNAL_ENGINE_ERROR".

11.4 FailureId → Wire Registry Code Mapping

11.4.1 Mapping table requirement

A conforming implementation MUST publish a mapping table:

```
FailureId -> WireErrorCodeId
```

This table MUST be:

- stable within a release,
- versioned,
- identical across implementations claiming equivalence under [WIRE]/[PROFILE].

11.4.2 Fallback rule (when Wire registry entry does not exist)

If no Wire registry ID exists for a given FailureId, the Analyzer MUST:

set ERROR.code to a stable token code:

"UNMAPPED_FAILURE"

and include the original FailureId in ERROR.details as:

```
details["binding.failure_id"] = FailureId
```

This preserves interoperability without inventing unofficial numeric IDs.

11.5 Normative Wire Code Ranges by Phase

For interoperability, the Analyzer MUST respect the [WIRE] registry grouping:

- VALIDATING → E1xxx
- EXPLORING → E2xxx
- EVALUATING → E3xxx
- ADAPTING → E4xxx
- GENERAL → E9xxx

If a mapping chooses a numeric Wire registry ID, it MUST be within the correct phase range.

11.6 Binding Mapping Table (Minimum Required Set)

The following mappings are REQUIRED in all conforming implementations.

11.6.1 Table: FailureId → WireErrorCodeId

FailureId	Wire phase	WireErrorCodeId
ANZ.VALIDATING.MALFORMED_REQUEST	VALIDATING	E1000 MALFORMED_REQUEST
ANZ.VALIDATING.MISSING_REQUIRED_FIELD	VALIDATING	E1001 MISSING_REQUIRED_FIELD
ANZ.VALIDATING.UNSUPPORTED_VERSION	VALIDATING	E1002 UNSUPPORTED_VERSION
ANZ.VALIDATING.UNKNOWN_REFERENCE	VALIDATING	E1003 UNKNOWN_CONTEXT_REFERENCE
ANZ.VALIDATING.CONTEXT_INCONSISTENT	VALIDATING	E1005 CONTEXT_INCONSISTENT
ENG.EXPLORING.EXPLORATION_BUDGET_EXCEEDED	EXPLORING	E2004 EXPLORATION_BUDGET_EXCEEDED
ENG.EVALUATING.CLASSIFICATION_UNDEFINED	EVALUATING	E3000 CLASSIFICATION_UNDEFINED
ENG.EVALUATING.COMPARABILITY_VIOLATION	EVALUATING	E3001 COMPARABILITY_VIOLATION
ENG.EVALUATING.DIMENSION_INVALID	EVALUATING	E3002 DIMENSION_INVALID
ENG.EVALUATING.INTERNAL_EVALUATION_ERROR	EVALUATING	E3003 INTERNAL_EVALUATION_ERROR
ENG.ADAPTING.PROGRESS_VIOLATION	ADAPTING	E4002 PROGRESS_VIOLATION
ANZ.GENERAL.INTERNAL_ERROR	GENERAL	E9000 INTERNAL_ERROR

Notes

If [WIRE] uses slightly different textual names for the code IDs, the numeric IDs remain authoritative.

The Analyzer MUST emit the ERROR.code exactly as required by [WIRE] (numeric id or the canonical token required by [WIRE]).

11.7 Mapping of Additional EngineErrorCode Values

11.7.1 Allowed extension

Implementations MAY define additional EngineError.code values. If so:

- They MUST define corresponding FailureId values.
- They MUST publish mapping table entries for each additional FailureId.
- If [WIRE] does not provide registry IDs, they MUST use the fallback rule (§11.4.2).

11.7.2 No ad-hoc numeric IDs

Implementations MUST NOT invent new numeric Exxxx codes beyond those provided by [WIRE] unless [WIRE] is explicitly extended and versioned.

11.8 Deterministic Error Selection (When multiple errors occur)

11.8.1 Single emitted error

Even if multiple failures occur internally, the Analyzer MUST emit exactly one ERROR.

11.8.2 Selection rule

When multiple fatal errors occur, the Analyzer MUST select the emitted error deterministically by the following priority:

- earliest state in execution order: VALIDATING < EXPLORING < EVALUATING < ADAPTING < GENERAL
- within same phase: lexicographically smallest FailureId (UTF-8 bytes)

The selected error MUST be disclosed as the emitted error.

Other errors MAY be recorded internally but MUST NOT affect output.

11.9 EngineErrorCode ↔ Profile Failure Identifiers

If [PROFILE] defines canonical analyzer-internal failure identifiers (tokens), the Analyzer MUST maintain a deterministic mapping:

```
EngineError.code -> ProfileFailureToken
```

Where:

ProfileFailureToken MUST be included in ERROR.details["profile.failure_token"] if available.

If no corresponding token exists: set profile.failure_token = "n/a".

This preserves traceability between this binding spec and [PROFILE].

11.10 Required Error Details (for audit)

Every emitted wire ERROR MUST include:

- details["binding.failure_id"] = FailureId

- details["binding.registry_snapshot_id"] = params.registry_snapshot_id (if available)
- details["binding.context_fingerprint"] = z.context_fingerprint.value (if available)
- details["binding.bind_params_fingerprint"] = bind_params_fingerprint.value (if available)

Details keys MUST be in canonical order.

12. Conformance Tests

This chapter defines the minimum required test suite for conformance to this binding specification. The test suite is normative: an implementation MUST pass all applicable tests for its claimed binding level.

All tests MUST be executable using only:

- the wire interface defined in [WIRE],
- the Analyzer behavior defined in [PROFILE], and
- the binding requirements defined in this document.

12.1 Test Execution Rules

12.1.1 Deterministic test environment [PROFILE:DCO]

All conformance tests MUST be executed under conditions that preserve determinism:

- fixed registry_snapshot_id,
- fixed BindParams (either explicit or fixed defaults),
- fixed canonicalization regime,
- no nondeterministic dependencies (time, randomness, locale, concurrency order).

12.1.2 Canonical test outputs [PROFILE:DCO]

Where [PROFILE] requires byte-level determinism (e.g., L2+), outputs MUST be compared after:

- canonical serialization per [PROFILE],
- and stable field ordering per this binding spec (§4.6, §9).

12.1.3 Required artifacts

A conforming implementation MUST provide:

- a test runner or procedure,
- the mapping tables required by §11,
- and a manifest stating supported binding levels (B1/B2/B3).

12.2 Required Test Inputs

12.2.1 Test carrier set

The test suite MUST include at minimum:

- C0: a valid carrier that canonicalizes successfully (EqClass exists)
- C_bad: an invalid carrier that fails canonicalization (malformed)
- C_edge: a valid carrier that triggers a boundary condition (e.g., minimal-size or maximal-size representative)
- C_cycle: a carrier/context that produces a detectable cycle in adaptation (B2/B3)

12.2.2 Test context set

The test suite MUST include at minimum:

- Z0: valid resolved context
- Z_partial: context with allow_partial=true
- Z_nopartial: context with allow_partial=false
- Z_bound: context with tight bounds (max_depth, max_steps, max_variants) to force truncation

All contexts MUST include stable fingerprints and declared coverage/transform properties.

12.2.3 Required BindParams sets

The suite MUST define at minimum:

- P_classify: run_mode=CLASSIFY_ONLY
- P_adapt: run_mode=ADAPT_IF_NEEDED
- P_force: run_mode=FORCE_ADAPT
- P_shallow: max_depth=0
- P_steps1: max_steps=1

Each test MUST state explicitly which BindParams set is used.

12.3 B1 Conformance Tests (Classification Binding)

T-B1-1: Request parsing and validation failure

Input: DIAGNOSE_REQUEST(C_bad, Z0)

Expected:

- run transitions to S6 FAILED
- emits exactly one ERROR
- ERROR.phase = VALIDATING
- ERROR.code = E1000 MALFORMED_REQUEST (or [WIRE]-equivalent)
- contains binding.failure_id

T-B1-2: Canonicalization determinism

Input: same request repeated N times (N≥5), DIAGNOSE_REQUEST(C0, Z0)

Expected:

- DIAGNOSTIC_FINDING.carrier_equivalence_id.class_representative_id identical for all runs
- canonical_carrier_hash identical
- bind_params_fingerprint identical
- all fields required by §9 present

T-B1-3: Profile completeness (no partial)

Input: DIAGNOSE_REQUEST(C0, Z_nopartial) with P_classify

Expected:

- emits DIAGNOSTIC_FINDING
- partial.allow_partial=false
- partial.is_partial=false
- all dimension_findings.valid=true
- ordering equals params.dim_order

T-B1-4: Finding comparability fingerprint stability

Input: DIAGNOSE_REQUEST(C0, Z0) repeated across two independent instances

Expected:

- context_fingerprint identical
- bind_params_fingerprint identical
- if [PROFILE] requires L2+ determinism: serialized finding bytes identical

12.4 B2 Conformance Tests (Adaptation Binding)

T-B2-1: Adaptation call binding and termination disclosure

Input: DIAGNOSE_REQUEST(C0, Z0) with P_adapt

Expected:

- run enters S3, S4, and may enter S4A
- emits DIAGNOSTIC_FINDING with termination.step_count ≥ 0
- includes adaptation.termination_reason
- includes exploration keys (§10.4.3) if exploration performed

T-B2-2: Max depth = 0 (Reach semantics)

Input: DIAGNOSE_REQUEST(C0, Z0) with P_shallow and exploration strategy REACH_BASED

Expected:

- exploration yields only {eq0}
- depth_reached = 0
- exploration.truncated=false unless additional bounds trigger truncation

T-B2-3: Max steps = 1

Input: DIAGNOSE_REQUEST(C0, Z0) with P_steps1

Expected:

- termination.step_count <= 1
- if adaptation is entered, termination reason is either:
- MAX_STEPS_REACHED, FIXPOINT, or CYCLE_DETECTED

disclosure includes binding.max_steps=1

T-B2-4: Pareto admissibility enforcement

Input: DIAGNOSE_REQUEST(C_edge, Z0) with P_force crafted such that:

at least one dimension update worsens while another improves (incomparable update)

Expected:

- such candidate MUST NOT be selected as eq_next
- selection MUST only be from Pareto maximal admissible set (§8.5)

T-B2-5: Deterministic selection

Input: DIAGNOSE_REQUEST(C0, Z0) with P_force, executed twice with same snapshot

Expected:

- identical selected trajectory EqClass sequence

- identical termination reason
- identical final finding bytes (if L2+)

T-B2-6: Cycle detection

Input: DIAGNOSE_REQUEST(C_cycle, Z0) with P_force

Expected:

- termination reason CYCLE_DETECTED
- disclosure includes adaptation.cycle_detected=true

12.5 Partial Completion Conformance Tests (B1/B2)

T-PART-1: allow_partial=true yields partial finding

Input: DIAGNOSE_REQUEST(C0, Z_partial) with a dimension crafted to fail evaluation deterministically

Expected:

- emits DIAGNOSTIC_FINDING
- partial.allow_partial=true
- partial.is_partial=true
- missing_dimension_ids lists the failing dimension(s)
- partial.reason_code = EVALUATION_FAILED (or equivalent)
- MUST NOT emit ERROR

T-PART-2: allow_partial=false yields ERROR

Input: same as T-PART-1 but Z_nopartial

Expected:

- emits ERROR
- ERROR.phase = EVALUATING
- code maps per §11 table or fallback
- MUST NOT emit DIAGNOSTIC_FINDING

12.6 B3 Conformance Tests Meta Binding [PROFILE:DCO]

T-B3-1: Meta finding emission

Input: any request where meta observation is performed (P_force + B3 enabled)

Expected:

- emits exactly one META_DIAGNOSTIC_FINDING
- meta_trajectory present and deterministically ordered

- disclosures include meta.enabled=true

T-B3-2: Meta determinism

Input: same request repeated twice with identical snapshot

Expected:

- identical meta_trajectory content
- identical theta_initial and theta_final disclosures

12.7 Error Mapping Conformance Tests All Levels [PROFILE:DCO]

T-ERR-1: Registry mapping completeness

Input: trigger each minimum required failure in §11.6

Expected:

- emitted ERROR.code matches required E#### values where defined
- otherwise emits UNMAPPED_FAILURE with binding.failure_id included

T-ERR-2: Deterministic multi-error selection

Input: crafted request producing multiple failures

Expected:

emitted error equals the deterministic selection rule (§11.8)

12.8 Test Reporting Requirements

12.8.1 Required report fields

A conformance report MUST include:

1. implementation identifier + version
2. supported binding levels (B1/B2/B3)
3. registry_snapshot_id used
4. all BindParams sets used
5. pass/fail per test case
6. hashes of canonical serialized outputs for determinism tests

12.8.2 Reproducibility

Reports MUST allow third-party reproduction:

- test vectors MUST be versioned
- canonical outputs MUST be hash-anchored