MODIVX STANDARD DEFINITION
WIRE OUTPUT ENVELOPE

Protocol version	1.0
Document version:	2.1
Status:	Release Candidate
Category:	Standards Track
Date:	2026-03
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 (Message Format and Serialization)

The Wire Output Envelope describes the specific structure and representation of the exchanged message (e.g., field names, field types, serialization format, layout conventions). Rationale for the requirement: Only after the semantics and protocol meaning have been clearly established can the binding 'packaging' be defined; a formal envelope without a clear framework of meaning leads to systematic misunderstandings.

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.

Principles

W-0.1 Determinism (MUST) [PROFILE:DCO]

For identical inputs (EqClass, Z, BindParams / Registry Snapshot), a conforming system MUST produce byte-identical normalized output (cf. STD-0.1 Normalizer).

W-0.2 Canonical Ordering (MUST) [PROFILE:DCO]

All sequences/sets in wire messages MUST be deterministically ordered:

- Dimensions: DimOrder
- EqClass sets: CanonKey-sorted
- Maps/dictionaries: lexicographically sorted by key, or deterministically serialized as sequences

W-0.3 Result Schema & Errors (MUST)

The wire layer transports:

- ordinal states strictly separated from error metadata (Errors / Confidence / Evidence),
- error codes as normalized enum strings (no “implicit error” via state degradation).

W-0.4 Disclosure (MUST)

Every wire response MUST disclose BindParams / Horizon plus registry_snapshot_id / config_fingerprint. [PROFILE:DCO]

1. Wire Message Universe

1.1 Message Container:

WireEnvelope

Every message is an object:

```
WireEnvelope := {
  "wire_version": String,          // MUST, e.g. "MLD-WIRE-2"
  "type": WireType,                // MUST
  "request_id": String,            // MUST, deterministic per request [PROFILE:DCO]
  "timestamp_utc": String,         // SHOULD, ISO-8601, if present deterministic source
  "payload": Object,               // MUST, type-specific schema
  "errors": [WireError]            // MUST (empty allowed), canonical ordered [PROFILE:DCO]
}
```

Requirements

- "wire_version" MUST = "MLD-WIRE-2" (or the exact version from Wire Protocol v2.0, if fixed)
- "type" MUST correspond to one of the types defined in 1.2
- "request_id" MUST be deterministically derivable from input (e.g., sha256 over normalized request) [PROFILE:DCO]
- "payload" MUST conform to the type schema
- "errors" MUST be present (empty list allowed)

If type = "ERROR", payload MUST NOT be present, and errors MUST contain exactly one element.

1.2 Wire Types (WireType)

```
WireType ∈ {
  "DIAG_REQUEST",
  "DIAG_RESULT",
  "ADAPT_REQUEST",
  "ADAPT_RESULT",
  "META_REQUEST",
  "META_RESULT",
  "CAPABILITIES_REQUEST",
  "CAPABILITIES_RESULT"
}
```

Note: Requests are optional in this standard package; the focus lies on deterministic outputs.

For conformance testing, run_case may directly generate result types.

The WireType values used in this document are normative representations of the message types defined in the “MLD-AWP Wire Protocol”; legacy type names are governed there as aliases.

The semantic meaning of DIAG_REQUEST and DIAG_RESULT is defined in Layer 3 (MODIVX Wire Protocol).

2. Shared Types

2.1 EqClassRef

```
EqClassRef := {
  "eqclass_key": String,          // MUST, CanonKey(Bytes) represented as String
  "format": String,               // MUST, e.g. "canon-bytes-b64" | "canon-hex"
  "label": String                 // OPTIONAL, human readable
}
```

Norms:

- "eqclass_key" MUST be deterministic [PROFILE:DCO]

- "format" MUST be specified
- "label", if present, is purely informative

2.2 ZRef (Context)

```
ZRef := {
  "z_id": String,                 // MUST, deterministic [PROFILE:DCO]
  "z_payload": Object             // OPTIONAL, must be serializable if present [PROFILE:DCO]
}
```

2.3 Dim / DimOrder

- Dim is transmitted as an Int or String (depending on the binding, but stable).
- DimOrder is transmitted as a list:

```
DimOrder := [Dim]   // MUST: each Dim exactly once
```

2.4 Horizon

```
Horizon := {
  "Nmax": Int,                    // MUST
  "CandidateBudget": Int,         // MUST
  "TimeBudget": Int               // MUST (0 allowed)
}
```

2.5 BindParams (Disclosure-Pflicht)

```
BindParams := {
  "dim_order": DimOrder,          // MUST
  "horizon": Horizon,             // MUST
  "approx_policy": String,        // SHOULD, e.g. "STD-0.1"
  "tie_break_policy": String,     // SHOULD, e.g. "NEUTRAL_SELECTION_RULE"
  "run_mode": String,             // OPTIONAL
  "max_steps": Int,               // REQUIRED for L3/L4 results
  "cycle_policy": String,         // REQUIRED for L3 results, e.g. "EQCLASS_KEY_REPEAT"
  "registry_snapshot_id": String, // MUST [PROFILE:DCO]
  "config_fingerprint": String    // OPTIONAL [PROFILE:DCO]
}
```

Normative rule:

registry_snapshot_id MUST be present; config_fingerprint MAY additionally be provided.

2.6 WireError

```
WireError := {
  "code": String,                 // MUST, normalized Code (siehe 2.6.1)
  "message": String,              // MUST, short
  "scope": String,                // MUST, e.g. "ENGINE" | "ANALYZER" | "WIRE"
  "details": Object               // OPTIONAL, serializable
}
```

2.6.1 Normierte Error Codes

```
WireError.code MUST ∈:
{
  "INVALID_INPUT",
  "UNSUPPORTED",
  "TIMEOUT",
  "EMPTY_SET",
  "NOT_CANONICAL",                          // [PROFILE:DCO]
  "NON_TERMINATING_GENERATOR",
  "INTERNAL_INCONSISTENCY",
  "INTERNAL_ERROR"
}
```

3. Diagnose (L2)

3.1 DIAG_RESULT Payload

```
DiagResultPayload := {
  "input": {
    "phi": EqClassRef,
    "z": ZRef
  },
  "bind_params": BindParams,

  "dimension_results": [DimDiagnosisBlock],     // MUST, DimOrder [PROFILE:DCO]
  "global_profile": Profile,                    // MUST
  "reactive_global": Bool,                      // MUST
  "stable_global": Bool,                        // MUST

  "disclosures": Disclosures                    // MUST
}
```

3.2 DimDiagnosisBlock

```
DimDiagnosisBlock := {
  "dim": Dim,                                   // MUST
  "delta": DiagnosisResult,                     // MUST
  "candidates": CandidateSetDisclosure          // MUST (may be minimal disclosure)
}
```

Ordering rule:

dimension_results MUST follow the order defined in bind_params.dim_order. [PROFILE:DCO]

3.3 DiagnosisResult

```
DiagnosisResult := {
  "state": StateLabel,              // MUST, ordinal label or code
  "confidence": ConfLabel,          // MUST: "LOW"|"MEDIUM"|"HIGH"
  "evidence": Evidence,             // MUST (empty allowed)
```


```
  "errors": [String]                    // MUST (empty allowed), canonical ordered [PROFILE:DCO]
}
```

State encoding:

state MUST be ordinal, not metric.

Permitted representations:

- StateLabel as string: "kritisch instabil" | "instabil" | …
- or StateLabel as int/enum code with disclosed mapping.

3.4 CandidateSetDisclosure

Two permitted modes exist:

1. Mode A (Full):

   ```
   CandidateSetDisclosure := {
   "mode": "FULL",
   "eqclasses": [EqClassRef]     // CanonKey sorted [PROFILE:DCO]
   }
   ```

2. Mode B (Hashed/IDs):

   ```
   CandidateSetDisclosure := {
   "mode": "HASHED",
   "count": Int,
   "fingerprints": [String]     // CanonKey sorted of fingerprints [PROFILE:DCO]
   }
   ```

Default:

For conformance, FULL is recommended; HASHED is permitted if deterministic. [PROFILE:DCO]

3.5 Profile

```
Profile := {
  "dims": [DimState]        // MUST, DimOrder [PROFILE:DCO]
}

DimState := { "dim": Dim, "state": StateLabel }
```

3.6 Disclosures

```
Disclosures := {
  "registry_snapshot_id": String,          // if present
  "config_fingerprint": String,            // if present [PROFILE:DCO]
  "transform_fingerprint": String,
  "context_fingerprint": String,
  "bind_params_fingerprint": String,       // [PROFILE:DCO]
```


```
  "horizon": Horizon,
  "approx_policy": String,
  "tie_break_policy": String,
  "canon_contract": CanonContractDisclosure,
  "rank_theta_direction": String
}
```

3.7 CanonContractDisclosure

```
CanonContractDisclosure := {
  "canon_format": String,                 // MUST, one of (“FULL”, “HASHED”)
  "transform_manifest_id": String,        // OPTIONAL, if present MUST correspond to
transform_fingerprint [PROFILE:DCO]
  "transform_fingerprint": String,        // deterministic fingerprint of the Binding Specs
[PROFILE:DCO]
  "max_collision_rate": Float,            // OPTIONAL
  "witness_set_id": String                // OPTIONAL
}
```

Normative rule:

If max_collision_rate is present and max_collision_rate > 0, then witness_set_id MUST be present.

4. Adaptation / Trajectory (L3)

4.1 ADAPT_RESULT Payload

```
AdaptResultPayload := {
  "input": {
    "phi0": EqClassRef,
    "z": ZRef,
    "theta": ThetaRef
  },
  "bind_params": BindParams,

  "trajectory": [EqClassRef],                    // MUST
  "step_logs": [StepLog],                        // MUST
  "stop_reason": StopReason,                     // MUST

  "final_profile": Profile,                      // MUST
  "stable_global_final": Bool,                   // MUST
  "reactive_global_final": Bool,                 // MUST

  "selection_disclosure": SelectionDisclosure    // MUST
  "disclosures": Disclosures                     // MUST
}
```

4.2 ThetaRef

```
ThetaRef := {
  "theta_id": String      // MUST
```

}

4.3 StopReason

StopReason ∈ { "FIXPOINT", "MAX_STEPS", "CYCLE_DETECTED" }

4.4 StepLog

Minimal normativer StepLog:

```
StepLog := {
  "n": Int,                          // MUST
  "phi_n": EqClassRef,               // MUST
  "i_n": Dim,                        // MUST
  "phi_n1": EqClassRef,              // MUST
  "theta": ThetaRef,                 // MUST
  "horizon": Horizon,                // MUST
  "admissible_count": Int,           // MUST
  "selection_trace": SelectionTrace  // MUST (may be minimal)
}
```

4.5 SelectionTrace (Neutral Selection Rule audit) [PROFILE:DCO]

```
SelectionTrace := {
  "stage": String,                   // MUST: "PARETO"|"EDITCOST"|"CANONKEY"
[PROFILE:DCO]
  "pareto_max_count": Int,           // MUST
  "editcost_min": Int,               // MUST (or +INF encoded)
  "canonkey_tiebreak_used": Bool     // MUST [PROFILE:DCO]
```

4.6 SelectionDisclosure [PROFILE:DCO]

```
SelectionDisclosure := {
  "neutral_selection_rule": Bool,          // MUST, true for L3
  "editcost_definition": String,           // MUST, e.g. "min_k s.t. psi in Reach_i_impl(phi,k)"
  "canonkey_only_final_tiebreak": Bool     // MUST
}
```

Ordering norms:

- trajectory MUST in the actual step order
- step_logs MUST increasing by n
- any candidate lists inside step logs must be CanonKey sorted if present

5. Meta Adaptation (L4)

5.1 META_RESULT Payload

```
MetaResultPayload := {
  "input": {
    "phi0": EqClassRef,
    "z": ZRef,
```

```
    "theta0": ThetaRef
  },
  "bind_params": BindParams,

  "theta_manifest": ThetaManifest,        // MUST
  "runs": [AdaptResultSummary],           // MUST
  "final_theta": ThetaRef,                // MUST

  "meta_observations": [MetaObs],         // MUST
  "meta_logs": [MetaLog],                 // MUST
  "meta_stop": MetaStopReason,            // MUST

  "disclosures": Disclosures              // MUST
}
```

5.2 ThetaManifest [PROFILE:DCO]

```
ThetaManifest := {
  "thetas": [ThetaRef],                   // MUST, deterministic order
  "rank_policy": String                   // MUST, describes RankTheta
}
```

5.3 AdaptResultSummary

```
AdaptResultSummary := {
  "theta": ThetaRef,
  "stop_reason": StopReason,
  "steps": Int,
  "fixpoint_reached": Bool,
  "cycle_detected": Bool
}
```

5.4 MetaObs [PROFILE:DCO]

```
MetaObs := {
  "obs_id": String,          // MUST, stable identifier from an Engine-declared observation-set
  "data": Object             // MUST, deterministic
}
```

5.5 MetaLog

```
MetaLog := {
  "t": Int,                          // MUST (meta iteration counter, MUST NOT be wall-clock time)
  "theta_in": ThetaRef,              // MUST
  "obs": MetaObs,                    // MUST
  "rank_in": Int,                    // MUST
  "theta_out": ThetaRef,             // MUST
  "rank_out": Int,                   // MUST
  "switch_accepted": Bool            // MUST
}
```

5.6 MetaStopReason [PROFILE:DCO]

MetaStopReason ∈ { "LOCKED", "FIXPOINT_REACHED", "THETA_EXHAUSTED" }

Normative constraints (must be checkable)

- rank_out MUST be monotonic in the declared RankTheta direction (see disclosure rank_theta_direction):
  - → if rank_theta_direction = NON_INCREASING, then rank_out <= rank_in
  - → if rank_theta_direction = NON_DECREASING, then rank_out >= rank_in
- once LOCKED is reached, no further switches are permitted

Clarification:

rank_in / rank_out is an internal deterministic selection index (meta-switching) and MUST NOT be interpreted as semantic profile ranking or aggregation of diagnostic profiles.

6. CAPABILITIES

6.1 CAPABILITIES_RESULT

```
CapabilitiesPayload := {
  "wire_version": String,
  "supported_levels": [String],          // e.g. ["L1","L2","L3"]
  "theta_manifest": ThetaManifest,       // if L4 supported
  "state_encoding": Object,              // disclosed mapping
  "error_codes": [String],               // list of supported codes
  "disclosures": Disclosures
}
```

7. Canonical Serialization Rules (wire-level) [PROFILE:DCO]

7.1 JSON Object ordering

If JSON is used:

- keys sorted lexicographically (Normalizer)
- no implicit maps with non-deterministic ordering

7.2 Sequence ordering

- Dim blocks: DimOrder. The semantics and canonical derivation of DimOrder are as specified in “MODIVX Implementation Profile”
- EqClass lists: CanonKey sort
- Errors: Since Errors MUST contain at most one element for a single run-level failure, no ordering rule applies to Errors.

7.3 Fingerprints (Binding-Kompatibilität)

The wire output MUST include the fingerprints required by the Binding Specification, in particular:

- context_fingerprint
- bind_param_fingerprint

Computation/definition of these fingerprints follows the Binding Spec (Required fingerprints / Disclosure).

8. Minimal Compliance Outputs

8.1 Minimal L1+L2 Output

A DIAG_RESULT is L1+L2 compliant if at minimum it includes:

1. WireEnvelope fields
2. bind_params with Horizon + dim_order
3. dimension_results in DimOrder
4. global_profile
5. disclosures

Terminal condition (normative):

A run MUST terminate with either a DIAG_RESULT or an ERROR.

For ERROR, WireEnvelope.type = "ERROR" and len(errors) = 1.

8.2 Minimal L3 Output

ADAPT_RESULT additionally requires:

1. trajectory + step_logs + stop_reason
2. selection_disclosure

8.3 Minimal L4 Output

META_RESULT additionally requires:

1. theta_manifest + runs + final_theta
2. meta_logs inkl.rank constraints

9. Conformance Hooks [PROFILE:DCO]

Every response MUST contain:

- bind_params
- disclosures
- context_fingerprint
- bind_params_fingerprint

so that conformance tests can verify that no hidden approximation is present.

Ownership rule:

The semantics of type names, mandatory fields, and fingerprints are defined exclusively in the referenced normative documents (Wire Protocol, Binding Specification, Schema). This document specifies only the wire representation.