# **Pre-Coding Essentials (Component: crates/vm\_io/src/loader.rs, Version/FormulaID: VM-ENGINE v0) — 34/89**

## **1) Goal & Success**

Goal: Load **local JSON artifacts** (manifest → registry → params → ballots *or* tally, optional adjacency), **validate** them against schemas, **normalize** ordering, and return a typed **LoadedContext** for the pipeline.

Success: Given a valid manifest, returns a fully-typed context with: IDs parsed, units/options sorted canonically, tallies/ballots shaped for tabulation, and early **referential checks** (unit/option IDs exist, reg\_id matches).

## **2) Scope**

In scope: File read, JSON parse, schema validation calls, ID parsing, canonical ordering, light referential checks, and construction of ephemeral types (UnitTallies/BallotsRaw, LoadedContext).

Out of scope: Heavy semantic validation (tree/root/magnitude rules, gates math), allocation/tabulation, report writing.

## **3) Inputs → Outputs**

Inputs: Paths from manifest::ResolvedPaths.

Outputs:

LoadedContext { reg, options, params, tally\_or\_ballots, adjacency\_inline?, ids }

Detected BallotSource (raw ballots vs tally).

## **4) Entities/Tables (minimal)**

Options are expected to be explicit (with order\_index) in the registry artifact; loader **requires** them for deterministic ordering.

## **5) Variables (loader knobs)**

## **6) Functions (signatures only)**

rust

CopyEdit

use crate::{IoError};

use vm\_core::{ids::\*, entities::\*, variables::Params};

pub enum TallyOrBallots { Tally(UnitTallies), Ballots(BallotsRaw) }

pub struct LoadedIds {

pub reg\_id: RegId,

pub param\_set\_id: ParamSetId,

pub tally\_id: Option<TallyId>, // None when raw ballots

}

pub struct LoadedContext {

pub reg: DivisionRegistry,

pub options: Vec<OptionItem>,

pub params: Params,

pub tally\_or\_ballots: TallyOrBallots,

pub adjacency\_inline: Option<Vec<Adjacency>>,

pub ids: LoadedIds,

}

// Top-level orchestration

pub fn load\_all\_from\_manifest(path: &std::path::Path) -> Result<LoadedContext, IoError>;

// Individual loaders (used by the above and by tests)

pub fn load\_registry(path: &std::path::Path) -> Result<(DivisionRegistry, Vec<OptionItem>), IoError>;

pub fn load\_params(path: &std::path::Path) -> Result<Params, IoError>;

pub fn load\_tally(path: &std::path::Path) -> Result<(TallyId, UnitTallies), IoError>;

pub fn load\_ballots(path: &std::path::Path) -> Result<BallotsRaw, IoError>;

// Cross-checks & normalization

pub fn normalize\_options(mut opts: Vec<OptionItem>) -> Vec<OptionItem>; // sort by (order\_index, id) + uniqueness checks

pub fn normalize\_units(mut units: Vec<Unit>) -> Vec<Unit>; // sort by UnitId

pub fn check\_cross\_refs(reg: &DivisionRegistry, opts: &[OptionItem], tally: &UnitTallies) -> Result<(), IoError>;

## **7) Algorithm Outline (implementation plan)**

**Orchestrate**

Read + parse manifest (manifest::load\_manifest), resolve paths, enforce expectations/digests.

Load registry → schema validate → parse IDs → extract options[] and units[] → normalize\_options/normalize\_units.

Load parameter set → schema validate → build Params (typed) → validate\_params (domain only).

**Choose source**

If tally path present: load tally → schema validate → parse TlyId.

Else: load raw ballots → schema validate → keep in BallotsRaw.

**Cross-checks (when tally)**

tally.reg\_id == reg.id (strict).

Every tally.units[i].unit\_id exists in registry.

Every option key referenced in tallies exists in options (by OptionId).

Option order\_index uniqueness and monotonicity (no duplicates, ≥1).

**Canonical ordering**

Sort units by UnitId ascending; options by (order\_index, OptionId); for each unit’s option maps, re-materialize as key-sorted structures (BTreeMap) to make downstream hashing independent of input order.

**Return**

Build LoadedIds { reg\_id, param\_set\_id, tally\_id? }; place adjacency list if registry contains it; return LoadedContext.

## **8) State Flow**

vm\_cli/vm\_pipeline calls load\_all\_from\_manifest → receives LoadedContext → pipeline runs **VALIDATE → TABULATE → …** using normalized, typed data.

## **9) Determinism & Numeric Rules**

Determinism via **stable sorts** and **key-sorted maps** before any hashing/serialization.

No floats parsed; counts remain integers.

No RNG.

## **10) Edge Cases & Failure Policy**

Missing/duplicate order\_index → **IoError::Manifest("option order\_index duplicate/missing")**.

Unknown unit\_id/OPT: in tallies → **IoError::Manifest("unknown unit/option id")**.

reg\_id mismatch between tally and registry → **IoError::Manifest("tally.reg\_id != registry.id")**.

Oversized file / parse depth exceeded → **IoError::Read/Json** with explicit limit names.

Raw ballots supplied: skip cross-checks that need tallies; pipeline will validate semantics after tabulation.

Loader never mutates counts; only sorts/normalizes structures.

## **11) Test Checklist (must pass)**

**Happy (tally):** Proper registry + params + tally → returns LoadedContext with sorted units/options and matching IDs.

**Happy (raw ballots):** Proper registry + params + ballots → returns context with TallyOrBallots::Ballots.

**Cross-ref failures:** unknown unit/option rejected; reg\_id mismatch rejected.

**Option ordering:** duplicates or out-of-domain order\_index rejected; equal index values break ties by OptionId only after uniqueness check.

**Determinism:** same inputs in permuted key orders → identical normalized memory layout (verified by serializing to canonical JSON and comparing hashes).