Annex A — Variable Canonical Reference Table.md

File

Annex B — Canonical Test Pack.md

File

Annex C — Glossary & Definitions.md

File

Doc 1 — Database Specification (Entities, Fields, Relationships).md

File

Doc 2 — Common Variables Specification (Core, Operational Defaults, Advanced Controls).md

File

Doc 3 — Technical Platform & Release Policy.md

File

Doc 4 — Algorithm Specification (Steps, Allocation, Gates & Edge Cases).md

File

Doc 5 — Processing Pipeline Specification (State Machine & Functions).md

File

Doc 6 — Test Specifications (Allocation, Gates, Frontier & Determinism).md

File

Doc 7 — Reporting Specification (Structure, Templates & Visual Rules).md

File

Here are the reference files . I will ask you to code specific files, aligned on this. Prepare yourself

Thought for 45s

**ChatGPT said:**

Got the 10 reference files. I’m ready. Below is the distilled alignment pack I’ll follow when coding any file you request.

**Coder’s Alignment Pack (concise)**

**Canonical data & identifiers**

* **Canonical JSON** everywhere (storage + hashing): UTF-8, LF newlines, sorted keys at all object levels; arrays have a fixed, spec-defined order. Artifact IDs are RES:, RUN:<UTC>-<hash>, and optional FR: computed from canonical bytes. FID is a SHA-256 over the Normative Manifest.
* **Entities / I/O triplet**: inputs = DivisionRegistry, BallotTally, ParameterSet; outputs = Result, RunRecord, optional FrontierMap; schemas and referential constraints are normative.

**Variables & FID membership (what changes outcomes)**

* **Included (in FID)**: ranges 001–007, 010–017, 020–031 (incl. 021, 029–031), 040–049, 050, 073. **Excluded**: 032–035, 052, 060–062. Tie policy **050** is Included; tie seed **052** is Excluded; **051** is reserved.
* Annex A is the **single source of truth** for each VM-VAR’s ID, domain, default, and FID inclusion; its manifest ordering and canonicalization rules govern FID construction.

**Determinism, RNG, and seeds (ties only)**

* No network, locale-neutral, time-agnostic; deterministic ordering and canonical JSON are required.
* RNG is **used only** when tie\_policy="random" and a tie actually occurs; seed = VM-VAR-052. Deterministic/deterministic\_order paths never draw RNG. The RNG profile is pinned by Annex B to guarantee cross-platform identity.

**Algorithm step order (what runs when)**

* Fixed pipeline: normalize/seed → per-unit tallies → **gates** (010–017, 020–031) → **frontier hook** (040–042, 047–049) → **allocation** (001–007, 073) → **tie resolution** (050/052) → aggregate + labels (060/061, presentation-only) → emit artifacts + FID.
* Deterministic ordering: units by ascending unit\_id; options within a unit by order\_index (then option\_id on equal).

**Engine pipeline & artifacts (state machine)**

* Canonical states from INIT/LOAD → VALIDATE → MANIFEST & SEED → per-unit loop (GATES → FRONTIER → ALLOCATE → TIES) → AGGREGATE & LABELS → BUILD ARTIFACTS → SELF-VERIFY → DONE, with defined error exits.
* RunRecord must echo input digests, effective vars, determinism block (tie\_policy, rng\_seed iff random ties), and tie events; artifacts must self-verify IDs/hashes before success.

**Tests & expected oracles**

* **Harness**: invoke CLI with the three canonical inputs; outputs must be canonical; FID recomputation must match Result/RunRecord. 6A forbids RNG use (no rng\_seed, ties=[]).
* **Annex B oracle**: expected/hashes.json is authoritative; when present, it overrides any “fat” expected files. It also pins tie expectations and the RNG profile.

**Reporting (renderer rules)**

* Renderer is **read-only** over Result, RunRecord, optional FrontierMap; it cannot recompute or reorder arrays. Presentation toggles 032–035, 060–062 control visibility/sort/layout only and are non-FID.
* Numeric display: percentages one decimal, round half up; shares three decimals; dates in UTC ISO; section structure and toggles are fixed.

**Glossary anchors (tokens I will use verbatim)**

* Canonical definitions for Algorithm family/variant, Frontier/FrontierMap, Gate, FID, Order index, Labels, etc., align to the current 050/052 scheme and non-FID presentation toggles.

**What I’ll assume unless you say otherwise (per-file work)**

1. **No network I/O**; outputs must be byte-identical across OS/arch for the same inputs + ParameterSet.
2. **Array order is canonical** (units by unit\_id, allocations by order\_index); never sort by votes in outputs.
3. **Tie handling** follows 4C: enforce tie\_policy 050; only log/seed when random. 051 is reserved.
4. **FID scope** uses Annex A’s Included set; presentation vars and seed 052 never affect FID.
5. **Tests**: I’ll target Annex-B hashes and 6A invariants by default unless you specify a 6B/6C case.

**When you give me a file to code**

Provide the file path/name and any deviations from defaults (e.g., enabling frontier, expected 6B/6C behaviors, renderer toggles). Otherwise I’ll implement to the specs above and verify against the canonical contracts and oracles.

Ready when you are.

Sources

**You said:**

23 - crates vm\_core src ids.rs.md

File

code this

Thought for 1m 18s

**ChatGPT said:**

Implementation for crates/vm\_core/src/ids.rs, aligned to the provided skeleton and canonical rules.

rust

CopyEdit

//! crates/vm\_core/src/ids.rs

//! Canonical engine/output IDs and token IDs (no input IDs here).

//! Deterministic, ASCII-only, strict shapes; no I/O.

#![allow(clippy::result\_large\_err)]

use core::fmt;

use core::str::FromStr;

#[cfg(feature = "serde")]

use serde::{Deserialize, Serialize};

/// Errors returned when validating or parsing IDs.

#[derive(Debug, Clone, Copy, PartialEq, Eq)]

pub enum IdError {

NonAscii,

TooLong,

BadShape,

}

const MAX\_ID\_LEN: usize = 256;

const HEX64\_LEN: usize = 64;

const TOKEN\_MAX\_LEN: usize = 64;

/// Quickly verify ASCII (no NUL).

#[inline]

fn is\_ascii\_no\_nul(s: &str) -> bool {

!s.as\_bytes().iter().any(|&b| b == 0 || b > 0x7F)

}

/// Lowercase hex (length must be exactly 64).

#[inline]

pub fn is\_valid\_sha256(s: &str) -> bool {

if s.len() != HEX64\_LEN || !is\_ascii\_no\_nul(s) {

return false;

}

s.as\_bytes()

.iter()

.all(|&b| (b'0'..=b'9').contains(&b) || (b'a'..=b'f').contains(&b))

}

/// Token for UnitId/OptionId: ^[A-Za-z0-9\_.:-]{1,64}$ (ASCII only)

#[inline]

pub fn is\_valid\_token(s: &str) -> bool {

let bs = s.as\_bytes();

let len = bs.len();

if len == 0 || len > TOKEN\_MAX\_LEN || !is\_ascii\_no\_nul(s) {

return false;

}

bs.iter().all(|&b| {

(b'A'..=b'Z').contains(&b)

|| (b'a'..=b'z').contains(&b)

|| (b'0'..=b'9').contains(&b)

|| b == b'\_'

|| b == b'.'

|| b == b':'

|| b == b'-'

})

}

macro\_rules! simple\_string\_newtype {

($(#[$m:meta])\* $name:ident) => {

$(#[$m])\*

#[derive(Clone, PartialEq, Eq, PartialOrd, Ord, Hash, Debug)]

#[cfg\_attr(feature = "serde", derive(Serialize, Deserialize))]

#[cfg\_attr(feature = "serde", serde(transparent))]

pub struct $name(String);

impl $name {

#[inline] pub fn as\_str(&self) -> &str { &self.0 }

}

impl fmt::Display for $name {

#[inline]

fn fmt(&self, f: &mut fmt::Formatter<'\_>) -> fmt::Result { f.write\_str(&self.0) }

}

impl TryFrom<&str> for $name {

type Error = IdError;

#[inline]

fn try\_from(value: &str) -> Result<Self, Self::Error> { value.parse() }

}

}

}

// === Hex-only newtypes: FormulaId, Sha256 ===

simple\_string\_newtype!(

/// 64-hex lowercase formula identifier (FID component).

FormulaId

);

simple\_string\_newtype!(

/// Generic 64-hex lowercase SHA-256 digest newtype.

Sha256

);

impl FromStr for FormulaId {

type Err = IdError;

fn from\_str(s: &str) -> Result<Self, Self::Err> {

if !is\_ascii\_no\_nul(s) { return Err(IdError::NonAscii); }

if s.len() > MAX\_ID\_LEN { return Err(IdError::TooLong); }

if !is\_valid\_sha256(s) { return Err(IdError::BadShape); }

Ok(FormulaId(s.to\_owned()))

}

}

impl FormulaId {

#[inline] pub fn as\_hex(&self) -> &str { &self.0 }

}

impl FromStr for Sha256 {

type Err = IdError;

fn from\_str(s: &str) -> Result<Self, Self::Err> {

if !is\_ascii\_no\_nul(s) { return Err(IdError::NonAscii); }

if s.len() > MAX\_ID\_LEN { return Err(IdError::TooLong); }

if !is\_valid\_sha256(s) { return Err(IdError::BadShape); }

Ok(Sha256(s.to\_owned()))

}

}

impl Sha256 {

#[inline] pub fn as\_hex(&self) -> &str { &self.0 }

}

// === Token IDs: UnitId, OptionId (no prefixes) ===

simple\_string\_newtype!(

/// Registry Unit token: ^[A-Za-z0-9\_.:-]{1,64}$

UnitId

);

simple\_string\_newtype!(

/// Registry Option token: ^[A-Za-z0-9\_.:-]{1,64}$

OptionId

);

impl FromStr for UnitId {

type Err = IdError;

fn from\_str(s: &str) -> Result<Self, Self::Err> {

if !is\_ascii\_no\_nul(s) { return Err(IdError::NonAscii); }

if s.len() > MAX\_ID\_LEN { return Err(IdError::TooLong); }

if !is\_valid\_token(s) { return Err(IdError::BadShape); }

Ok(UnitId(s.to\_owned()))

}

}

impl FromStr for OptionId {

type Err = IdError;

fn from\_str(s: &str) -> Result<Self, Self::Err> {

if !is\_ascii\_no\_nul(s) { return Err(IdError::NonAscii); }

if s.len() > MAX\_ID\_LEN { return Err(IdError::TooLong); }

if !is\_valid\_token(s) { return Err(IdError::BadShape); }

Ok(OptionId(s.to\_owned()))

}

}

// === Prefixed output IDs: RES, RUN, FR ===

simple\_string\_newtype!(

/// "RES:" + 64-hex lowercase

ResultId

);

simple\_string\_newtype!(

/// "RUN:" + <RFC3339 UTC 'YYYY-MM-DDTHH:MM:SSZ'> + "-" + 64-hex lowercase

RunId

);

simple\_string\_newtype!(

/// "FR:" + 64-hex lowercase

FrontierMapId

);

#[inline]

fn is\_res\_shape(s: &str) -> bool {

s.len() == 4 + HEX64\_LEN

&& s.as\_bytes().get(0..4) == Some(b"RES:")

&& is\_valid\_sha256(&s[4..])

}

#[inline]

fn is\_fr\_shape(s: &str) -> bool {

s.len() == 3 + HEX64\_LEN

&& s.as\_bytes().get(0..3) == Some(b"FR:")

&& is\_valid\_sha256(&s[3..])

}

/// Strict RFC3339 "YYYY-MM-DDTHH:MM:SSZ"

#[inline]

fn is\_rfc3339\_utc\_20(ts: &str) -> bool {

let b = ts.as\_bytes();

if b.len() != 20 { return false; }

// YYYY-MM-DDTHH:MM:SSZ

let digits = |r: core::ops::Range<usize>| b[r].iter().all(|&c| (b'0'..=b'9').contains(&c));

digits(0..4)

&& b[4] == b'-'

&& digits(5..7)

&& b[7] == b'-'

&& digits(8..10)

&& b[10] == b'T'

&& digits(11..13)

&& b[13] == b':'

&& digits(14..16)

&& b[16] == b':'

&& digits(17..19)

&& b[19] == b'Z'

}

#[inline]

fn is\_run\_shape(s: &str) -> bool {

// "RUN:" + ts(20) + "-" + hex64

if s.len() != 4 + 20 + 1 + HEX64\_LEN { return false; }

let b = s.as\_bytes();

if b.get(0..4) != Some(b"RUN:") { return false; }

let ts = &s[4..24];

if !is\_rfc3339\_utc\_20(ts) { return false; }

if b[24] != b'-' { return false; }

is\_valid\_sha256(&s[25..])

}

impl FromStr for ResultId {

type Err = IdError;

fn from\_str(s: &str) -> Result<Self, Self::Err> {

if !is\_ascii\_no\_nul(s) { return Err(IdError::NonAscii); }

if s.len() > MAX\_ID\_LEN { return Err(IdError::TooLong); }

if !is\_res\_shape(s) { return Err(IdError::BadShape); }

Ok(ResultId(s.to\_owned()))

}

}

impl ResultId {

#[inline] pub fn as\_hex(&self) -> &str { &self.0[4..] }

}

impl FromStr for FrontierMapId {

type Err = IdError;

fn from\_str(s: &str) -> Result<Self, Self::Err> {

if !is\_ascii\_no\_nul(s) { return Err(IdError::NonAscii); }

if s.len() > MAX\_ID\_LEN { return Err(IdError::TooLong); }

if !is\_fr\_shape(s) { return Err(IdError::BadShape); }

Ok(FrontierMapId(s.to\_owned()))

}

}

impl FrontierMapId {

#[inline] pub fn as\_hex(&self) -> &str { &self.0[3..] }

}

impl FromStr for RunId {

type Err = IdError;

fn from\_str(s: &str) -> Result<Self, Self::Err> {

if !is\_ascii\_no\_nul(s) { return Err(IdError::NonAscii); }

if s.len() > MAX\_ID\_LEN { return Err(IdError::TooLong); }

if !is\_run\_shape(s) { return Err(IdError::BadShape); }

Ok(RunId(s.to\_owned()))

}

}

impl RunId {

/// Fast accessor to the embedded timestamp (RFC3339 UTC).

#[inline]

pub fn timestamp\_utc(&self) -> &str {

// "RUN:" + <ts 20> + "-" + hex64

&self.0[4..24]

}

}

// === Tests ===

#[cfg(test)]

mod tests {

use super::\*;

#[test]

fn sha\_and\_formula() {

let hex = "0123456789abcdef0123456789abcdef0123456789abcdef0123456789abcd";

assert!(is\_valid\_sha256(hex));

let fid: FormulaId = hex.parse().unwrap();

assert\_eq!(fid.as\_hex(), hex);

let dig: Sha256 = hex.parse().unwrap();

assert\_eq!(format!("{dig}"), hex);

assert!("0123XYZ".parse::<Sha256>().is\_err());

}

#[test]

fn tokens() {

for ok in ["A", "a", "9", "\_", ".", ":", "-", "A\_b:9.Z"] {

assert!(is\_valid\_token(ok));

let \_u: UnitId = ok.parse().unwrap();

let \_o: OptionId = ok.parse().unwrap();

}

for bad in ["", " ", "é", "toolong\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_"] {

assert!(!is\_valid\_token(bad));

assert!(bad.parse::<UnitId>().is\_err());

}

}

#[test]

fn res\_fr\_run() {

let hex = "0123456789abcdef0123456789abcdef0123456789abcdef0123456789abcd";

let res\_s = format!("RES:{hex}");

let fr\_s = format!("FR:{hex}");

let run\_s = format!("RUN:2025-08-12T14:00:00Z-{hex}");

let res: ResultId = res\_s.parse().unwrap();

let fr: FrontierMapId = fr\_s.parse().unwrap();

let run: RunId = run\_s.parse().unwrap();

assert\_eq!(res.as\_hex(), hex);

assert\_eq!(fr.as\_hex(), hex);

assert\_eq!(run.timestamp\_utc(), "2025-08-12T14:00:00Z");

// Round-trip

assert\_eq!(format!("{res}"), res\_s);

assert\_eq!(format!("{fr}"), fr\_s);

assert\_eq!(format!("{run}"), run\_s);

// Bad shapes

assert!("RES:DEADBEAF".parse::<ResultId>().is\_err());

assert!("FR:0123XYZ...".parse::<FrontierMapId>().is\_err());

assert!("RUN:2025-08-12T14:00:00-0123".parse::<RunId>().is\_err()); // missing Z and '-'

assert!("RUN:2025-08-12 14:00:00Z-".to\_string() + hex

.as\_str()).parse::<RunId>().is\_err(); // space instead of 'T'

}

}

Sources

**You said:**

24 - crates vm\_core src entities.rs.md

File

code this

Thought for 46s

**ChatGPT said:**

Implementation for crates/vm\_core/src/entities.rs, aligned to the provided skeleton (24/89).

rust

CopyEdit

//! crates/vm\_core/src/entities.rs

//! Domain entities shared across the engine (registry, units, options, tallies).

//! Pure types + invariants + deterministic ordering helpers. No I/O.

#![allow(clippy::result\_large\_err)]

use core::cmp::Ordering;

use core::fmt;

use crate::ids::{OptionId, UnitId};

#[cfg(feature = "serde")]

use serde::{Deserialize, Serialize};

/// Construction/validation errors for domain entities.

#[derive(Debug, Clone, Copy, PartialEq, Eq)]

pub enum EntityError {

EmptyCollection,

InvalidName,

}

impl fmt::Display for EntityError {

fn fmt(&self, f: &mut fmt::Formatter<'\_>) -> fmt::Result {

match self {

EntityError::EmptyCollection => f.write\_str("empty collection"),

EntityError::InvalidName => f.write\_str("invalid name"),

}

}

}

#[cfg(feature = "std")]

impl std::error::Error for EntityError {}

const NAME\_MIN\_LEN: usize = 1;

const NAME\_MAX\_LEN: usize = 200;

#[inline]

fn is\_valid\_name(s: &str) -> bool {

let len = s.chars().count();

(NAME\_MIN\_LEN..=NAME\_MAX\_LEN).contains(&len)

}

/// Canonical registry of divisions/units/options.

/// Invariant: `units.len() >= 1`, units kept (or sortable) in ↑ UnitId order.

#[derive(Clone, Debug, PartialEq, Eq, Hash)]

#[cfg\_attr(feature = "serde", derive(Serialize, Deserialize))]

pub struct DivisionRegistry {

pub schema\_version: String,

pub units: Vec<Unit>,

}

impl DivisionRegistry {

/// Construct a registry, enforcing non-empty & per-unit invariants.

pub fn new(schema\_version: String, mut units: Vec<Unit>) -> Result<Self, EntityError> {

if units.is\_empty() {

return Err(EntityError::EmptyCollection);

}

// Validate units & their options

for u in &units {

u.assert\_invariants()?;

}

// Canonicalize order

sort\_units\_by\_id(&mut units);

Ok(Self { schema\_version, units })

}

/// Read-only view of all units.

#[inline]

pub fn units(&self) -> &[Unit] {

&self.units

}

/// Find a unit by id (linear scan; call after canonical sort if you prefer binary\_search).

#[inline]

pub fn unit(&self, id: &UnitId) -> Option<&Unit> {

self.units.iter().find(|u| &u.unit\_id == id)

}

}

/// A voting/geographic unit (e.g., district, ward).

/// Invariant: name: 1..=200 chars; `options.len() >= 1`; options kept in ↑(order\_index, option\_id).

#[derive(Clone, Debug, PartialEq, Eq, Hash)]

#[cfg\_attr(feature = "serde", derive(Serialize, Deserialize))]

pub struct Unit {

pub unit\_id: UnitId,

pub name: String,

pub protected\_area: bool,

pub options: Vec<OptionItem>,

}

impl Unit {

pub fn new(

unit\_id: UnitId,

name: String,

protected\_area: bool,

mut options: Vec<OptionItem>,

) -> Result<Self, EntityError> {

if !is\_valid\_name(&name) {

return Err(EntityError::InvalidName);

}

if options.is\_empty() {

return Err(EntityError::EmptyCollection);

}

for o in &options {

o.assert\_invariants()?;

}

sort\_options\_canonical(&mut options);

Ok(Self {

unit\_id,

name,

protected\_area,

options,

})

}

/// Root-ness is a pipeline concern; keep API but return false to avoid leaking policy here.

#[inline]

pub fn is\_root(&self) -> bool {

false

}

#[inline]

fn assert\_invariants(&self) -> Result<(), EntityError> {

if !is\_valid\_name(&self.name) {

return Err(EntityError::InvalidName);

}

if self.options.is\_empty() {

return Err(EntityError::EmptyCollection);

}

for o in &self.options {

o.assert\_invariants()?;

}

Ok(())

}

}

/// An option/party/candidate entry within a unit.

/// Invariant: name 1..=200 chars; order\_index is a u16; canonical cmp is (order\_index, option\_id).

#[derive(Clone, Debug, PartialEq, Eq, Hash)]

#[cfg\_attr(feature = "serde", derive(Serialize, Deserialize))]

pub struct OptionItem {

pub option\_id: OptionId,

pub name: String,

pub order\_index: u16,

}

impl OptionItem {

pub fn new(option\_id: OptionId, name: String, order\_index: u16) -> Result<Self, EntityError> {

if !is\_valid\_name(&name) {

return Err(EntityError::InvalidName);

}

Ok(Self {

option\_id,

name,

order\_index,

})

}

#[inline]

fn assert\_invariants(&self) -> Result<(), EntityError> {

if !is\_valid\_name(&self.name) {

return Err(EntityError::InvalidName);

}

Ok(())

}

}

/// Per-unit totals, mirroring the BallotTally aggregation.

#[derive(Clone, Copy, Debug, PartialEq, Eq, Hash, Default)]

#[cfg\_attr(feature = "serde", derive(Serialize, Deserialize))]

pub struct TallyTotals {

pub valid\_ballots: u64,

pub invalid\_ballots: u64,

}

impl TallyTotals {

#[inline]

pub fn new(valid\_ballots: u64, invalid\_ballots: u64) -> Self {

Self {

valid\_ballots,

invalid\_ballots,

}

}

/// Total ballots cast (saturating to guard against theoretical overflow).

#[inline]

pub fn ballots\_cast(&self) -> u64 {

self.valid\_ballots.saturating\_add(self.invalid\_ballots)

}

}

/// Deterministic canonical comparison for options: ↑ (order\_index, option\_id).

#[inline]

pub fn cmp\_options(a: &OptionItem, b: &OptionItem) -> Ordering {

match a.order\_index.cmp(&b.order\_index) {

Ordering::Equal => a.option\_id.cmp(&b.option\_id),

ord => ord,

}

}

/// Sort units by UnitId ascending (stable).

#[inline]

pub fn sort\_units\_by\_id(units: &mut [Unit]) {

units.sort\_by(|a, b| a.unit\_id.cmp(&b.unit\_id));

}

/// Sort options canonically by (order\_index, option\_id) (stable).

#[inline]

pub fn sort\_options\_canonical(opts: &mut [OptionItem]) {

opts.sort\_by(cmp\_options);

}

#[cfg(test)]

mod tests {

use super::\*;

fn uid(s: &str) -> UnitId {

s.parse().unwrap()

}

fn oid(s: &str) -> OptionId {

s.parse().unwrap()

}

#[test]

fn name\_bounds() {

assert!(is\_valid\_name("A"));

assert!(is\_valid\_name(&"x".repeat(200)));

assert!(!is\_valid\_name(""));

assert!(!is\_valid\_name(&"x".repeat(201)));

}

#[test]

fn option\_new\_and\_invariants() {

let ok = OptionItem::new(oid("A"), "Alpha".into(), 0).unwrap();

assert\_eq!(ok.order\_index, 0);

assert!(OptionItem::new(oid("B"), "".into(), 1).is\_err());

}

#[test]

fn unit\_new\_and\_sorting() {

let o3 = OptionItem::new(oid("Z"), "Zed".into(), 2).unwrap();

let o1 = OptionItem::new(oid("A"), "Alpha".into(), 0).unwrap();

let o2 = OptionItem::new(oid("B"), "Bravo".into(), 1).unwrap();

let mut u = Unit::new(uid("U1"), "Unit".into(), false, vec![o3.clone(), o2.clone(), o1.clone()]).unwrap();

// Canonical order check: (order\_index, option\_id)

assert\_eq!(u.options[0].option\_id, oid("A"));

assert\_eq!(u.options[1].option\_id, oid("B"));

assert\_eq!(u.options[2].option\_id, oid("Z"));

// Bad name

assert!(Unit::new(uid("U2"), "".into(), false, vec![o1.clone()]).is\_err());

// Empty options

assert!(Unit::new(uid("U3"), "Ok".into(), false, vec![]).is\_err());

}

#[test]

fn registry\_new\_and\_lookup() {

let u1 = Unit::new(uid("A"), "A".into(), false, vec![OptionItem::new(oid("X"), "X".into(), 1).unwrap()]).unwrap();

let u2 = Unit::new(uid("B"), "B".into(), true, vec![OptionItem::new(oid("Y"), "Y".into(), 0).unwrap()]).unwrap();

let reg = DivisionRegistry::new("1.0".into(), vec![u2.clone(), u1.clone()]).unwrap();

// Sorted by unit\_id

assert\_eq!(reg.units[0].unit\_id, uid("A"));

assert\_eq!(reg.units[1].unit\_id, uid("B"));

// Lookup

assert!(reg.unit(&uid("A")).is\_some());

assert!(reg.unit(&uid("Z")).is\_none());

}

#[test]

fn totals\_sum\_is\_saturating() {

let t = TallyTotals::new(u64::MAX, 10);

assert\_eq!(t.ballots\_cast(), u64::MAX); // saturates, never panics

}

#[test]

fn cmp\_and\_sort\_options() {

let mut v = vec![

OptionItem::new(oid("B"), "B".into(), 0).unwrap(),

OptionItem::new(oid("A"), "A".into(), 0).unwrap(),

OptionItem::new(oid("C"), "C".into(), 1).unwrap(),

];

sort\_options\_canonical(&mut v);

assert\_eq!(v.iter().map(|o| o.option\_id.as\_str()).collect::<Vec<\_>>(), ["A", "B", "C"]);

}

}

Sources

**You said:**

28 - crates vm\_core src rng.rs.md

File

code this.

Thought for 54s

**ChatGPT said:**

Implementation for crates/vm\_core/src/rng.rs, aligned to the skeleton (28/89).

rust

CopyEdit

//! crates/vm\_core/src/rng.rs

//! Deterministic RNG for tie resolution only (VM-VAR-052 seed).

//! ChaCha20-based, single-threaded, no OS entropy/time, no floats.

#![allow(clippy::result\_large\_err)]

use rand\_chacha::ChaCha20Rng;

use rand\_core::{RngCore, SeedableRng};

use smol\_str::SmolStr;

#[cfg(feature = "serde")]

use serde::{Deserialize, Serialize};

/// Construction/usage errors (kept for API symmetry; most fns return Option on empty domains).

#[derive(Debug, Clone, Copy, PartialEq, Eq)]

pub enum RngError {

EmptyDomain,

}

/// Opaque RNG used strictly for ties. Tracks how many 64-bit words were consumed.

#[derive(Clone)]

pub struct TieRng {

rng: ChaCha20Rng,

words\_consumed: u128,

}

/// Optional audit crumb (callers may aggregate into RunRecord tie logs).

#[derive(Debug, Clone, PartialEq, Eq)]

#[cfg\_attr(feature = "serde", derive(Serialize, Deserialize))]

pub struct TieCrumb {

pub ctx: SmolStr,

pub pick: u32,

pub word\_index: u128,

}

/// Build from integer tie\_seed (VM-VAR-052). Stable across platforms.

#[inline]

pub fn tie\_rng\_from\_seed(seed: u64) -> TieRng {

TieRng {

rng: ChaCha20Rng::seed\_from\_u64(seed),

words\_consumed: 0,

}

}

impl TieRng {

/// Next unbiased integer in [0, n) via rejection sampling. Returns None if n == 0.

#[inline]

pub fn gen\_range(&mut self, n: u64) -> Option<u64> {

if n == 0 {

return None;

}

// Avoid modulo bias with rejection sampling.

// zone is the largest multiple of n that fits in u64.

let zone = u64::MAX - (u64::MAX % n);

loop {

let x = self.next\_u64();

if x < zone {

return Some(x % n);

}

}

}

/// Choose index of winner from slice; None on empty slice.

#[inline]

pub fn choose\_index<T>(&mut self, slice: &[T]) -> Option<usize> {

let n = slice.len() as u64;

self.gen\_range(n).map(|v| v as usize)

}

/// Deterministic in-place Fisher–Yates shuffle.

#[inline]

pub fn shuffle<T>(&mut self, xs: &mut [T]) {

let len = xs.len();

if len <= 1 {

return;

}

// Iterate i = len-1 down to 1, choose j ∈ [0, i]

for i in (1..len).rev() {

// unwrap is safe: i >= 1 ⇒ i+1 >= 2 ⇒ domain non-zero

let j = self.gen\_range((i as u64) + 1).unwrap() as usize;

xs.swap(i, j);

}

}

/// Emit next u64.

#[inline]

pub fn next\_u64(&mut self) -> u64 {

let v = self.rng.next\_u64();

self.words\_consumed = self.words\_consumed.saturating\_add(1);

v

}

/// Emit next u128 (concat of two u64 draws).

#[inline]

pub fn next\_u128(&mut self) -> u128 {

let hi = self.next\_u64() as u128;

let lo = self.next\_u64() as u128;

(hi << 64) | lo

}

/// Return how many 64-bit words have been consumed.

#[inline]

pub fn words\_consumed(&self) -> u128 {

self.words\_consumed

}

/// Optional: build a tiny crumb for audit logs (does not consume RNG).

#[inline]

pub fn log\_pick(&self, ctx: &str, pick: usize) -> TieCrumb {

TieCrumb {

ctx: SmolStr::new(ctx),

pick: pick as u32,

word\_index: self.words\_consumed(),

}

}

}

#[cfg(test)]

mod tests {

use super::\*;

#[test]

fn determinism\_same\_seed\_same\_sequence() {

let mut a = tie\_rng\_from\_seed(42);

let mut b = tie\_rng\_from\_seed(42);

let mut va = [0u64; 8];

let mut vb = [0u64; 8];

for i in 0..8 {

va[i] = a.next\_u64();

vb[i] = b.next\_u64();

}

assert\_eq!(va, vb);

// gen\_range determinism

let mut a = tie\_rng\_from\_seed(7);

let mut b = tie\_rng\_from\_seed(7);

let sa: Vec<u64> = (0..20).map(|\_| a.gen\_range(10).unwrap()).collect();

let sb: Vec<u64> = (0..20).map(|\_| b.gen\_range(10).unwrap()).collect();

assert\_eq!(sa, sb);

}

#[test]

fn different\_seeds\_diverge() {

let mut a = tie\_rng\_from\_seed(1);

let mut b = tie\_rng\_from\_seed(2);

// Very likely to diverge immediately

let x = a.next\_u64();

let y = b.next\_u64();

assert\_ne!(x, y);

}

#[test]

fn choose\_and\_shuffle() {

let mut r = tie\_rng\_from\_seed(123);

// choose\_index

let empty: [i32; 0] = [];

assert!(r.choose\_index(&empty).is\_none());

let v = [10, 20, 30];

let idx = r.choose\_index(&v).unwrap();

assert!(idx < v.len());

// shuffle determinism

let mut a = (0..10).collect::<Vec<\_>>();

let mut b = (0..10).collect::<Vec<\_>>();

let mut r1 = tie\_rng\_from\_seed(9);

let mut r2 = tie\_rng\_from\_seed(9);

r1.shuffle(&mut a);

r2.shuffle(&mut b);

assert\_eq!(a, b);

// different seed ⇒ almost surely different permutation

let mut c = (0..10).collect::<Vec<\_>>();

let mut r3 = tie\_rng\_from\_seed(99);

r3.shuffle(&mut c);

assert\_ne!(a, c);

}

#[test]

fn words\_and\_u128() {

let mut r = tie\_rng\_from\_seed(5);

assert\_eq!(r.words\_consumed(), 0);

let \_u = r.next\_u64();

assert\_eq!(r.words\_consumed(), 1);

let \_v = r.next\_u128();

assert\_eq!(r.words\_consumed(), 3);

}

#[test]

fn crumb\_does\_not\_consume\_words() {

let mut r = tie\_rng\_from\_seed(77);

let \_ = r.next\_u64();

let before = r.words\_consumed();

let c = r.log\_pick("tie#1", 2);

assert\_eq!(c.ctx.as\_str(), "tie#1");

assert\_eq!(c.pick, 2);

assert\_eq!(c.word\_index, before);

assert\_eq!(r.words\_consumed(), before);

}

#[test]

fn gen\_range\_zero\_none() {

let mut r = tie\_rng\_from\_seed(0);

assert\_eq!(r.gen\_range(0), None);

}

}

Sources

**You said:**

29 - crates vm\_io Cargo.toml.md

File

code this

Thought for 1m 19s

**ChatGPT said:**

Cargo manifest for crates/vm\_io/Cargo.toml, aligned to the 29/89 skeleton.

toml

CopyEdit

[package]

name = "vm\_io"

version = "0.1.0"

edition = "2021"

rust-version = "1.77"

license = "Apache-2.0 OR MIT"

description = "I/O, canonical JSON, JSON Schema validation (2020-12), and SHA-256 hashing for the VM engine."

# repository = "."; readme = "README.md"

[lib]

name = "vm\_io"

path = "src/lib.rs"

crate-type = ["rlib"]

[features]

default = ["std", "serde", "schemaval", "hash", "path\_utf8"]

std = []

serde = ["dep:serde", "dep:serde\_json"]

schemaval = ["dep:jsonschema"]

hash = ["dep:sha2", "dep:digest"]

path\_utf8 = ["dep:camino"]

[dependencies]

vm\_core = { path = "../vm\_core" } # adjust if your workspace layout differs

serde = { version = "1", features = ["derive"], optional = true, default-features = false }

serde\_json = { version = "1", optional = true } # std only; used for parse/write

jsonschema = { version = "0.17", optional = true, default-features = false, features = ["draft2020-12"] }

sha2 = { version = "0.10", optional = true, default-features = false }

digest = { version = "0.10", optional = true, default-features = false }

hex = { version = "0.4", default-features = false }

camino = { version = "1", optional = true, default-features = false }

thiserror = { version = "1", default-features = false }

[dev-dependencies]

assert\_json\_diff = "2"

tempfile = "3"

Notes: matches the goals/scope and feature layout in the skeleton; defaults are additive and off-by-default where possible, with optional gates for serde, schema validation, hashing, and UTF-8 paths.

Sources

**You said:**

31 - crates vm\_io src canonical\_json.rs.md

File

code this. aligned on the 10 reference doc i gave you . do you still access them ? Always do for each file

Thought for 2m 24s

**ChatGPT said:**

Implementation for crates/vm\_io/src/canonical\_json.rs, aligned to the skeleton (31/89) and the 10 reference docs (canonical JSON, determinism, and artifact rules).

rust

CopyEdit

//! crates/vm\_io/src/canonical\_json.rs

//! Canonical JSON bytes (UTF-8; objects with lexicographically sorted keys; arrays untouched).

//! Compact (default) and pretty (2-space, LF) variants; atomic file write.

//!

//! Notes:

//! - Keys are sorted by UTF-8 byte order using a recursive transformer.

//! - Compact: minimal whitespace, no trailing newline.

//! - Pretty: 2-space indent, LF newlines (no trailing newline).

//! - Errors bubble from serde\_json/std::io (assumes crate::IoError has From conversions).

use std::collections::BTreeMap;

use std::fs::{self, File, OpenOptions};

use std::io::{BufWriter, Write};

use std::path::{Path, PathBuf};

use serde::Serialize;

use serde\_json::{self as sj, Map, Value};

use crate::IoError;

/// Return canonical JSON bytes (UTF-8, sorted keys, compact; no trailing newline).

pub fn to\_canonical\_bytes<T: Serialize>(value: &T) -> Result<Vec<u8>, IoError> {

// 1) Serialize to Value (serde\_json already rejects NaN/Inf)

let v: Value = sj::to\_value(value).map\_err(IoError::from)?;

// 2) Recursively sort object keys; arrays untouched

let v = canonicalize\_value(v);

// 3) Emit compact bytes (no trailing newline)

let mut out = Vec::with\_capacity(4096);

let mut ser = sj::Serializer::new(&mut out);

v.serialize(&mut ser).map\_err(IoError::from)?;

Ok(out)

}

/// Pretty variant (2-space indent) that still sorts keys and enforces LF.

pub fn to\_canonical\_pretty\_bytes<T: Serialize>(value: &T) -> Result<Vec<u8>, IoError> {

let v: Value = sj::to\_value(value).map\_err(IoError::from)?;

let v = canonicalize\_value(v);

let mut out = Vec::with\_capacity(4096);

let fmt = sj::ser::PrettyFormatter::with\_indent(b" "); // PrettyFormatter emits '\n' (LF)

let mut ser = sj::Serializer::with\_formatter(&mut out, fmt);

v.serialize(&mut ser).map\_err(IoError::from)?;

Ok(out)

}

/// Write canonical JSON file (creates parent dirs; atomic replace via temp+rename).

pub fn write\_canonical\_file<T: Serialize, P: AsRef<Path>>(value: &T, path: P) -> Result<(), IoError> {

let path = path.as\_ref();

// Build bytes

let bytes = to\_canonical\_bytes(value)?;

// Ensure parent exists

if let Some(parent) = path.parent() {

fs::create\_dir\_all(parent).map\_err(IoError::from)?;

}

// Temp path: "<file>.<ext>.tmp" or "<file>.tmp"

let tmp\_path = make\_tmp\_path(path);

// Write to temp, flush, fsync

{

let f = OpenOptions::new()

.create(true)

.write(true)

.truncate(true)

.open(&tmp\_path)

.map\_err(IoError::from)?;

let mut w = BufWriter::new(f);

w.write\_all(&bytes).map\_err(IoError::from)?;

w.flush().map\_err(IoError::from)?;

// Ensure durability before rename

w.into\_inner().map\_err(IoError::from)?.sync\_all().map\_err(IoError::from)?;

}

// Atomic-ish replace: try rename; if the target exists on platforms where rename doesn't replace,

// remove and retry (best-effort cross-platform).

match fs::rename(&tmp\_path, path) {

Ok(()) => Ok(()),

Err(e) => {

// Attempt best-effort replace

let \_ = fs::remove\_file(path);

fs::rename(&tmp\_path, path).map\_err(IoError::from).or\_else(|\_| {

// Final fallback: copy+remove (not strictly atomic but prevents dangling temp files)

let mut f = File::create(path).map\_err(IoError::from)?;

f.write\_all(&bytes).map\_err(IoError::from)?;

let \_ = fs::remove\_file(&tmp\_path);

Err(IoError::from(e))

})?;

Ok(())

}

}

}

/// Recursively sort all JSON object keys using BTreeMap; arrays/scalars untouched.

fn canonicalize\_value(v: Value) -> Value {

match v {

Value::Object(map) => {

// First canonicalize children

let mut bt: BTreeMap<String, Value> = BTreeMap::new();

for (k, v) in map {

bt.insert(k, canonicalize\_value(v));

}

// Rebuild in sorted order

let mut out: Map<String, Value> = Map::with\_capacity(bt.len());

for (k, v) in bt {

out.insert(k, v);

}

Value::Object(out)

}

Value::Array(xs) => {

// Preserve array order (caller is responsible for deterministic ordering upstream)

Value::Array(xs.into\_iter().map(canonicalize\_value).collect())

}

other => other,

}

}

fn make\_tmp\_path(path: &Path) -> PathBuf {

match path.extension().and\_then(|e| e.to\_str()) {

Some(ext) if !ext.is\_empty() => {

let mut s = ext.to\_string();

s.push\_str(".tmp");

let mut p = path.to\_path\_buf();

p.set\_extension(s);

p

}

\_ => {

let mut p = path.to\_path\_buf();

p.set\_extension("tmp");

p

}

}

}

#[cfg(test)]

mod tests {

use super::\*;

use serde::Deserialize;

use std::str;

#[derive(Serialize, Deserialize, PartialEq, Debug)]

struct Demo {

a: u32,

m: std::collections::HashMap<String, u32>,

v: Vec<u8>,

}

#[test]

fn ordering\_and\_idempotence\_compact() {

let mut m1 = std::collections::HashMap::new();

m1.insert("z".into(), 1);

m1.insert("a".into(), 2);

let d1 = Demo { a: 1, m: m1, v: vec![3, 2, 1] };

let b1 = to\_canonical\_bytes(&d1).unwrap();

// Reparse & reserialize → identical

let parsed: Value = sj::from\_slice(&b1).unwrap();

let b2 = to\_canonical\_bytes(&parsed).unwrap();

assert\_eq!(b1, b2);

// Keys sorted

let s = str::from\_utf8(&b1).unwrap();

assert!(s.find("\"a\":1").is\_some());

assert!(s.find("\"m\":{").is\_some());

assert!(s.find("\"a\":2").is\_some()); // inside "m", "a" comes before "z"

}

#[test]

fn pretty\_has\_only\_whitespace\_differences() {

let mut m1 = std::collections::BTreeMap::new();

m1.insert("b".to\_string(), 1u32);

m1.insert("a".to\_string(), 2u32);

let v = serde\_json::json!({ "k": m1, "arr": [3,2,1] });

let c = to\_canonical\_bytes(&v).unwrap();

let p = to\_canonical\_pretty\_bytes(&v).unwrap();

// Parse both; semantic equality

let vc: Value = sj::from\_slice(&c).unwrap();

let vp: Value = sj::from\_slice(&p).unwrap();

assert\_eq!(vc, vp);

// Pretty contains LF

assert!(str::from\_utf8(&p).unwrap().contains('\n'));

}

#[test]

fn arrays\_untouched() {

let v = serde\_json::json!({ "x": [ {"b":1,"a":2}, {"d":4,"c":3} ]});

let out = to\_canonical\_bytes(&v).unwrap();

let s = str::from\_utf8(&out).unwrap();

// Inner objects have sorted keys, array order preserved

assert!(s.contains(r#"{"a":2,"b":1}"#));

assert!(s.contains(r#"{"c":3,"d":4}"#));

assert!(s.find(r#"{"a":2,"b":1}"#) < s.find(r#"{"c":3,"d":4}"#));

}

}

**Why this matches the refs:** Canonical JSON with sorted keys and stable formatting is a determinism primitive; any change to these rules is normative and requires a new FID (Doc 3B §3/§6). The pipeline reads/writes only canonical artifacts and must be byte-identical across OS/arch (Doc 5A §§1–2, §6). Renderer never mutates canonical JSON (Doc 7A §1–§3).

Yes—I have and will keep using the 10 reference docs for every file you ask me to code.

Sources

**You said:**

32 - crates vm\_io src manifest.rs.md

File

code this

Thought for 2m 10s

**ChatGPT said:**

Implementation for crates/vm\_io/src/manifest.rs, aligned to the skeleton and the normative refs (offline, determinism, pipeline inputs).

rust

CopyEdit

//! crates/vm\_io/src/manifest.rs

//! Parse, validate, and resolve the run \*\*manifest.json\*\* into concrete local paths,

//! with optional expectations and digest verification. No network I/O.

//!

//! Contracts (concise):

//! - Required paths: registry, params, \*\*ballot\_tally\*\*. Optional: adjacency. (Doc 5A S0)

//! - Offline only: reject URLs; resolve relative paths against the manifest's directory. (Doc 3A §3)

//! - Optional `expect.{formula\_id,engine\_version}` and `digests{}` checks.

use std::collections::BTreeMap;

use std::fs;

use std::io::{Error as IoStdError, ErrorKind};

use std::path::{Path, PathBuf};

use camino::{Utf8Path, Utf8PathBuf};

use serde::{Deserialize, Serialize};

use serde\_json::{self as sj, Value};

#[cfg(feature = "schemaval")]

use jsonschema::{Draft, JSONSchema};

use sha2::{Digest, Sha256};

use crate::IoError;

// ---------- Module knobs (can be surfaced as Config later) ----------

const REJECT\_URLS: bool = true;

const ALLOW\_PARENT\_TRAVERSAL: bool = true; // set false to confine inside base\_dir

const MAX\_BYTES: usize = 2 \* 1024 \* 1024;

// Schema embedded from crate root: <crate>/schemas/manifest.schema.json

#[cfg(feature = "schemaval")]

const MANIFEST\_SCHEMA\_JSON: &str = include\_str!("../schemas/manifest.schema.json");

// ---------- Public API types ----------

#[derive(Debug, Clone, Serialize, Deserialize)]

pub struct Manifest {

pub id: String, // "MAN:…"

pub reg\_path: String,

pub params\_path: String,

pub ballot\_tally\_path: String, // REQUIRED (normative pipeline)

pub adjacency\_path: Option<String>,

pub expect: Option<Expect>,

pub digests: Option<BTreeMap<String, DigestEntry>>,

pub notes: Option<String>,

}

#[derive(Debug, Clone, Serialize, Deserialize)]

pub struct Expect {

pub formula\_id: Option<String>, // expected 64-hex

pub engine\_version: Option<String>, // semver-ish string

}

#[derive(Debug, Clone, Serialize, Deserialize)]

pub struct DigestEntry {

pub sha256: String, // 64-hex (lowercase preferred)

}

#[derive(Debug, Clone)]

pub struct ResolvedPaths {

pub base\_dir: Utf8PathBuf,

pub reg: Utf8PathBuf,

pub params: Utf8PathBuf,

pub tally: Utf8PathBuf,

pub adjacency: Option<Utf8PathBuf>,

}

// ---------- Top-level functions ----------

pub fn load\_manifest<P: AsRef<Path>>(path: P) -> Result<Manifest, IoError> {

let path = path.as\_ref();

// Size guard + read

let meta = fs::metadata(path).map\_err(IoError::from)?;

if meta.len() as usize > MAX\_BYTES {

return Err(invalid\_data\_err(format!(

"manifest exceeds MAX\_BYTES ({} > {})",

meta.len(),

MAX\_BYTES

)));

}

let bytes = fs::read(path).map\_err(IoError::from)?;

// Parse JSON

let raw: Value = sj::from\_slice(&bytes).map\_err(|e| json\_err("/", e))?;

// Defensive: reject legacy `ballots\_path` if present

if let Some(obj) = raw.as\_object() {

if obj.contains\_key("ballots\_path") {

return Err(invalid\_data\_err("legacy field `ballots\_path` is not allowed".into()));

}

}

// Optional: schema validation

#[cfg(feature = "schemaval")]

{

let schema\_v: Value = sj::from\_str(MANIFEST\_SCHEMA\_JSON)

.map\_err(|e| invalid\_data\_err(format!("invalid embedded manifest.schema.json: {e}")))?;

let compiled = JSONSchema::options()

.with\_draft(Draft::Draft2020\_12)

.compile(&schema\_v)

.map\_err(|e| invalid\_data\_err(format!("schema compile error: {e}")))?;

if let Err(iter) = compiled.validate(&raw) {

// Report first violation with a JSON Pointer-like path

if let Some(err) = iter.into\_iter().next() {

let ptr = err.instance\_path.to\_string();

let msg = err.to\_string();

return Err(schema\_err(

if ptr.is\_empty() { "/" } else { ptr.as\_str() },

msg,

));

}

}

}

// Into typed Manifest

let man: Manifest = sj::from\_value(raw).map\_err(|e| json\_err("/", e))?;

Ok(man)

}

pub fn validate\_manifest(man: &Manifest) -> Result<(), IoError> {

// Required: ballot\_tally\_path non-empty

if man.ballot\_tally\_path.trim().is\_empty() {

return Err(invalid\_data\_err("`ballot\_tally\_path` is required and must be non-empty".into()));

}

// URL rejection (all path-like fields)

if REJECT\_URLS {

for (label, s) in [

("reg\_path", &man.reg\_path),

("params\_path", &man.params\_path),

("ballot\_tally\_path", &man.ballot\_tally\_path),

] {

if looks\_like\_url(s) {

return Err(invalid\_data\_err(format!("{} must be a local file path (no URLs)", label)));

}

}

if let Some(adj) = &man.adjacency\_path {

if looks\_like\_url(adj) {

return Err(invalid\_data\_err("adjacency\_path must be a local file path (no URLs)".into()));

}

}

}

// Optional digests: quick shape validation (hex form only; file existence checked in verify\_digests)

if let Some(dmap) = &man.digests {

for (k, v) in dmap {

if !["reg\_path", "params\_path", "ballot\_tally\_path", "adjacency\_path"].contains(&k.as\_str()) {

return Err(invalid\_data\_err(format!("unknown digest key `{}`", k)));

}

if !is\_hex64(&v.sha256) {

return Err(invalid\_data\_err(format!("digest for `{}` must be 64 hex chars", k)));

}

}

}

Ok(())

}

pub fn resolve\_paths<P: AsRef<Path>>(manifest\_file: P, man: &Manifest) -> Result<ResolvedPaths, IoError> {

let base\_dir\_fs: &Path = manifest\_file

.as\_ref()

.parent()

.ok\_or\_else(|| invalid\_data\_err("manifest\_file has no parent directory".into()))?;

let base\_dir\_utf8 = to\_utf8(base\_dir\_fs)?;

let reg = normalize\_join(&base\_dir\_utf8, &man.reg\_path)?;

let params = normalize\_join(&base\_dir\_utf8, &man.params\_path)?;

let tally = normalize\_join(&base\_dir\_utf8, &man.ballot\_tally\_path)?;

let adjacency = match &man.adjacency\_path {

Some(p) => Some(normalize\_join(&base\_dir\_utf8, p)?),

None => None,

};

// Optional confinement: disallow escaping base\_dir

if !ALLOW\_PARENT\_TRAVERSAL {

for (label, p) in [

("reg\_path", &reg),

("params\_path", &params),

("ballot\_tally\_path", &tally),

] {

if !p.starts\_with(&base\_dir\_utf8) {

return Err(invalid\_data\_err(format!(

"{} escapes base\_dir after normalization",

label

)));

}

}

if let Some(adj) = &adjacency {

if !adj.starts\_with(&base\_dir\_utf8) {

return Err(invalid\_data\_err("adjacency\_path escapes base\_dir after normalization".into()));

}

}

}

Ok(ResolvedPaths {

base\_dir: base\_dir\_utf8,

reg,

params,

tally,

adjacency,

})

}

pub fn enforce\_expectations(

man: &Manifest,

engine\_version: &str,

formula\_id\_hex: &str,

) -> Result<(), IoError> {

if let Some(exp) = &man.expect {

if let Some(exp\_fid) = &exp.formula\_id {

if !is\_hex64(exp\_fid) {

return Err(expect\_err("expect.formula\_id is not a 64-hex string"));

}

if exp\_fid.to\_ascii\_lowercase() != formula\_id\_hex.to\_ascii\_lowercase() {

return Err(expect\_err("formula\_id mismatch"));

}

}

if let Some(exp\_eng) = &exp.engine\_version {

if exp\_eng != engine\_version {

return Err(expect\_err("engine\_version mismatch"));

}

}

}

Ok(())

}

pub fn verify\_digests(

paths: &ResolvedPaths,

digests: &BTreeMap<String, DigestEntry>,

) -> Result<(), IoError> {

for (k, v) in digests {

let (label, p) = match k.as\_str() {

"reg\_path" => ("registry", &paths.reg),

"params\_path" => ("params", &paths.params),

"ballot\_tally\_path" => ("ballot\_tally", &paths.tally),

"adjacency\_path" => {

let adj = paths

.adjacency

.as\_ref()

.ok\_or\_else(|| invalid\_data\_err("digest provided for adjacency\_path, but manifest has no adjacency\_path".into()))?;

("adjacency", adj)

}

\_ => return Err(invalid\_data\_err(format!("unknown digest path: {}", k))),

};

// Read bytes and compute sha256

let bytes = fs::read(p).map\_err(IoError::from)?;

let mut hasher = Sha256::new();

hasher.update(&bytes);

let got = hex::encode(hasher.finalize());

if got.to\_ascii\_lowercase() != v.sha256.to\_ascii\_lowercase() {

return Err(invalid\_data\_err(format!(

"digest mismatch for {}: expected {}, got {}",

label, v.sha256, got

)));

}

}

Ok(())

}

// ---------- Helpers ----------

fn json\_err(pointer: &str, e: sj::Error) -> IoError {

invalid\_data\_err(format!("JSON parse error at {}: {}", pointer, e))

}

fn schema\_err(pointer: &str, msg: String) -> IoError {

invalid\_data\_err(format!("Schema violation at {}: {}", pointer, msg))

}

fn invalid\_data\_err(msg: impl Into<String>) -> IoError {

IoError::from(IoStdError::new(ErrorKind::InvalidData, msg.into()))

}

fn expect\_err(msg: impl Into<String>) -> IoError {

IoError::from(IoStdError::new(ErrorKind::InvalidData, format!("Expect: {}", msg.into())))

}

fn is\_hex64(s: &str) -> bool {

if s.len() != 64 {

return false;

}

s.bytes().all(|b| (b'0'..=b'9').contains(&b) || (b'a'..=b'f').contains(&b) || (b'A'..=b'F').contains(&b))

}

fn looks\_like\_url(s: &str) -> bool {

let sl = s.trim().to\_ascii\_lowercase();

sl.starts\_with("http://") || sl.starts\_with("https://")

}

fn to\_utf8(p: &Path) -> Result<Utf8PathBuf, IoError> {

Utf8Path::from\_path(p)

.map(|u| u.to\_owned())

.ok\_or\_else(|| invalid\_data\_err(format!("non-UTF8 path: {}", p.display())))

}

// Lexical normalization (no filesystem access): resolves "." and ".." components.

// This does not follow symlinks and keeps absolute vs relative-ness.

fn lexical\_normalize(mut p: Utf8PathBuf) -> Utf8PathBuf {

let is\_abs = p.is\_absolute();

let mut out = Utf8PathBuf::new();

for comp in p.components() {

match comp.as\_str() {

"." => {}

".." => {

let \_ = out.pop();

}

other => out.push(other),

}

}

if is\_abs {

// Ensure we keep absolute if original was absolute

if !out.is\_absolute() {

out = Utf8Path::new("/").join(out);

}

}

out

}

fn normalize\_join(base: &Utf8Path, rel: &str) -> Result<Utf8PathBuf, IoError> {

// If `rel` is absolute, take as-is; else join to base.

let relp = Utf8Path::new(rel);

let joined = if relp.is\_absolute() {

relp.to\_path\_buf()

} else {

base.join(relp)

};

Ok(lexical\_normalize(joined))

}

// ---------- Tests ----------

#[cfg(test)]

mod tests {

use super::\*;

use std::fs::File;

use std::io::Write;

use tempfile::tempdir;

#[test]

fn expect\_checks() {

let man = Manifest {

id: "MAN:demo".into(),

reg\_path: "reg.json".into(),

params\_path: "params.json".into(),

ballot\_tally\_path: "tally.json".into(),

adjacency\_path: None,

expect: Some(Expect {

formula\_id: Some("0123456789abcdef0123456789abcdef0123456789abcdef0123456789abcd".into()),

engine\_version: Some("v1.2.3".into()),

}),

digests: None,

notes: None,

};

// OK

enforce\_expectations(

&man,

"v1.2.3",

"0123456789abcdef0123456789abcdef0123456789abcdef0123456789abcd",

)

.unwrap();

// Bad hex

let mut man\_bad = man.clone();

man\_bad.expect.as\_mut().unwrap().formula\_id = Some("xyz".into());

assert!(enforce\_expectations(&man\_bad, "v1.2.3", "00").is\_err());

// Mismatch

let mut man\_bad2 = man.clone();

man\_bad2.expect.as\_mut().unwrap().engine\_version = Some("v9.9.9".into());

assert!(enforce\_expectations(

&man\_bad2,

"v1.2.3",

"0123456789abcdef0123456789abcdef0123456789abcdef0123456789abcd"

)

.is\_err());

}

#[test]

fn resolve\_and\_verify\_digests() {

let dir = tempdir().unwrap();

let base = dir.path().join("cases");

fs::create\_dir\_all(&base).unwrap();

// Write three files

let reg\_p = base.join("reg.json");

let params\_p = base.join("params.json");

let tally\_p = base.join("tally.json");

for (p, content) in [

(&reg\_p, br#"{"a":1}"#.as\_slice()),

(&params\_p, br#"{"b":2}"#.as\_slice()),

(&tally\_p, br#"{"c":3}"#.as\_slice()),

] {

let mut f = File::create(p).unwrap();

f.write\_all(content).unwrap();

}

let man = Manifest {

id: "MAN:demo".into(),

reg\_path: "reg.json".into(),

params\_path: "params.json".into(),

ballot\_tally\_path: "tally.json".into(),

adjacency\_path: None,

expect: None,

digests: None,

notes: None,

};

let man\_path = base.join("manifest.json");

fs::write(&man\_path, br#"{"id":"x","reg\_path":"reg.json","params\_path":"params.json","ballot\_tally\_path":"tally.json"}"#).unwrap();

let res = resolve\_paths(&man\_path, &man).unwrap();

// Compute digests and verify

let mut d = BTreeMap::new();

for (k, p) in [

("reg\_path", res.reg.clone()),

("params\_path", res.params.clone()),

("ballot\_tally\_path", res.tally.clone()),

] {

let bytes = fs::read(p).unwrap();

let mut h = Sha256::new();

h.update(&bytes);

d.insert(k.to\_string(), DigestEntry { sha256: hex::encode(h.finalize()) });

}

verify\_digests(&res, &d).unwrap();

}

#[test]

fn url\_rejection\_and\_legacy() {

// URL rejection via validate\_manifest

let man = Manifest {

id: "MAN:demo".into(),

reg\_path: "https://example.com/reg.json".into(),

params\_path: "params.json".into(),

ballot\_tally\_path: "tally.json".into(),

adjacency\_path: None,

expect: None,

digests: None,

notes: None,

};

assert!(validate\_manifest(&man).is\_err());

}

}

**Notes on alignment**

* Inputs/outputs and manifest’s role map to **Doc 5A S0 INIT & LOAD**; no network I/O; paths are local-only.
* Offline & determinism constraints (no URLs/network, byte-identical runs) follow **Doc 3A §§2–3**.
* The required **ballot\_tally** path and typed resolution match the skeleton’s contract for this file.

Sources

**You said:**

35 - crates vm\_algo Cargo.toml.md

File

code this

Thought for 47s

**ChatGPT said:**

crates/vm\_algo/Cargo.toml — aligned to the 10 reference docs (algo only; no I/O/JSON/UI; determinism via vm\_core).

toml

CopyEdit

[package]

name = "vm\_algo"

version = "0.1.0"

edition = "2021"

rust-version = "1.77"

license = "Apache-2.0 OR MIT"

description = "Algorithm layer: tabulation, seat allocation, gates/frontier; depends only on vm\_core."

# repository = "..."; readme = "README.md"

[lib]

name = "vm\_algo"

path = "src/lib.rs"

crate-type = ["rlib"]

[features]

# Keep this crate algorithmic-only; features gate families without changing public types.

default = ["std", "tab\_ranked", "tab\_score", "pr\_methods", "mmp", "gates", "frontier"]

std = []

tab\_ranked = []

tab\_score = []

pr\_methods = []

mmp = []

gates = []

frontier = []

[dependencies]

vm\_core = { path = "../vm\_core" } # RNG and rounding come from vm\_core

[dev-dependencies]

proptest = "1" # property tests

rand\_chacha = "0.3" # tests only; runtime RNG is vm\_core::rng

Notes: single dependency on vm\_core; no transitive JSON/schema/FS/UI. Feature flags only gate compilation scope (ranked/score/PR/MMP/gates/frontier) and do not alter public types.

Sources

**You said:**

36 - crates vm\_algo src lib.rs.md

File

code this

Thought for 2m 8s

**ChatGPT said:**

Implementation for crates/vm\_algo/src/lib.rs, aligned to your 10 reference docs (Algo 4A–4C, Pipeline 5A, Tests 6A–6C, Reporting 7A) and the skeleton you provided.

rust

CopyEdit

//! crates/vm\_algo/src/lib.rs

//! Public surface for pure algorithm primitives (tabulation, allocation, gates/frontier).

//! No I/O, no JSON; deterministic ordering; RNG only for ties per 050/052.

//!

//! Ordering rules (normative reminders):

//! - Units iterate by ascending `unit\_id`; options by `(order\_index, option\_id)`; allocations mirror registry order. :contentReference[oaicite:4]{index=4}

//! - RNG is used only for random tie policy (050) and seeded from 052 by the pipeline. :contentReference[oaicite:5]{index=5}

#![cfg\_attr(not(feature = "std"), no\_std)]

extern crate alloc;

use alloc::collections::BTreeMap;

use alloc::vec::Vec;

pub use vm\_core::{

ids::{OptionId, UnitId},

rng::TieRng,

// For API clarity, alias vm\_core's TallyTotals as Turnout here.

entities::TallyTotals as Turnout,

// Ratio type (integer rational compare/formatting) lives in vm\_core.

rounding::Ratio,

// Variable enums (tie policy, allocation method, etc.) are defined in vm\_core::variables.

variables::{AllocationMethod, OverhangPolicy, TiePolicy, TotalSeatsModel},

// Registry option metadata (order\_index) used for canonical ordering.

entities::OptionItem,

};

/// Raw scores per unit, ready for allocation/aggregation.

#[derive(Clone, Debug, PartialEq, Eq)]

pub struct UnitScores {

pub unit\_id: UnitId,

pub turnout: Turnout, // valid/invalid/total ballots

pub scores: BTreeMap<OptionId, u64>, // plurality=votes; approval=approvals; score=score sums

}

/// Per-unit allocation result; deterministic ordering by (order\_index, OptionId).

#[derive(Clone, Debug, PartialEq, Eq)]

pub struct Allocation {

pub unit\_id: UnitId,

/// For PR methods: seats per option. For WTA single-member: winner may be represented as 1 seat

/// (pipeline interprets accordingly). Ordering mirrors registry option order. :contentReference[oaicite:6]{index=6}

pub seats\_or\_power: BTreeMap<OptionId, u32>,

/// True iff a tie policy decided a last seat / winner. (Pipeline logs tie details.) :contentReference[oaicite:7]{index=7}

pub last\_seat\_tie: bool,

}

/// IRV round transfer/audit data (minimal, engine-agnostic).

#[derive(Clone, Debug, PartialEq, Eq)]

pub struct IrvRound {

pub eliminated: OptionId,

pub transfers: BTreeMap<OptionId, u64>,

pub exhausted: u64,

}

/// IRV log across rounds.

#[derive(Clone, Debug, PartialEq, Eq)]

pub struct IrvLog {

pub rounds: Vec<IrvRound>,

pub winner: OptionId,

}

/// Pairwise wins map for Condorcet completions.

#[derive(Clone, Debug, PartialEq, Eq)]

pub struct Pairwise {

/// (A,B) = votes preferring A over B.

pub wins: BTreeMap<(OptionId, OptionId), u64>,

}

/// Gate outcome (ratios are integer rationals; no floats).

#[derive(Clone, Copy, Debug, PartialEq, Eq)]

pub struct GateOutcome {

pub pass: bool,

pub observed: Ratio,

pub threshold\_pct: u8,

}

/// Double-majority composition (e.g., national + regional).

#[derive(Clone, Copy, Debug, PartialEq, Eq)]

pub struct DoubleMajority {

pub national: GateOutcome,

pub regional: GateOutcome,

pub pass: bool,

}

// ---- Module layout (stubs). Implementations live in sibling modules. ---------------------------

// The pipeline calls these in the step order defined in Doc 4A/5A; tests conform to Doc 6A–6C.

// (We provide function signatures here with `unimplemented!()` to make the public API explicit.)

/// Tabulation primitives (plurality/approval/score, IRV, Condorcet).

pub mod tabulation {

use super::\*;

/// Plurality: raw vote counts per option for a unit.

pub fn tabulate\_plurality(

unit\_id: UnitId,

votes: &BTreeMap<OptionId, u64>,

turnout: Turnout,

) -> UnitScores {

let \_ = (unit\_id, votes, turnout);

unimplemented!("tabulate\_plurality (4A S1)"); // :contentReference[oaicite:8]{index=8}

}

/// Approval: approvals per option; denominator for shares is valid\_ballots. :contentReference[oaicite:9]{index=9}

pub fn tabulate\_approval(

unit\_id: UnitId,

approvals: &BTreeMap<OptionId, u64>,

turnout: Turnout,

) -> UnitScores {

let \_ = (unit\_id, approvals, turnout);

unimplemented!("tabulate\_approval (4A S1)");

}

/// Score: sums of scores per option. Scale domain validated upstream; integer-only here. :contentReference[oaicite:10]{index=10}

pub fn tabulate\_score(

unit\_id: UnitId,

score\_sums: &BTreeMap<OptionId, u64>,

turnout: Turnout,

) -> UnitScores {

let \_ = (unit\_id, score\_sums, turnout);

unimplemented!("tabulate\_score (4A S1)");

}

/// IRV on compressed ranked ballots; fixed exhaustion policy per spec; deterministic ties via 050/052. :contentReference[oaicite:11]{index=11}

pub fn tabulate\_ranked\_irv(

ballots: &[(Vec<OptionId>, u64)], // unique options per ranking; multiplicity

options: &[OptionItem], // ordered by (order\_index, id)

) -> (UnitScores, IrvLog) {

let \_ = (ballots, options);

unimplemented!("tabulate\_ranked\_irv (4A S1 + 4C ties)");

}

/// Condorcet: pairwise tallies + completion; deterministic ordering. :contentReference[oaicite:12]{index=12}

pub fn tabulate\_ranked\_condorcet(

ballots: &[(Vec<OptionId>, u64)],

options: &[OptionItem],

) -> (UnitScores, Pairwise) {

let \_ = (ballots, options);

unimplemented!("tabulate\_ranked\_condorcet (4A S1)");

}

}

/// Allocation methods within a unit (WTA, divisors, largest remainder).

pub mod allocation {

use super::\*;

/// Winner-take-all for single-member magnitude. Ties via policy 050; RNG only if Random (052). :contentReference[oaicite:13]{index=13}

pub fn allocate\_wta(

scores: &UnitScores,

magnitude: u32,

options: &[OptionItem],

tie\_policy: TiePolicy,

mut rng: Option<&mut TieRng>,

) -> Allocation {

let \_ = (scores, magnitude, options, tie\_policy, &mut rng);

unimplemented!("allocate\_wta (4A S4 + 4C ties)");

}

/// D’Hondt (Jefferson). Deterministic order per registry options. :contentReference[oaicite:14]{index=14}

pub fn allocate\_dhondt(

seats: u32,

scores: &BTreeMap<OptionId, u64>,

options: &[OptionItem],

) -> BTreeMap<OptionId, u32> {

let \_ = (seats, scores, options);

unimplemented!("allocate\_dhondt (4A S4)");

}

/// Sainte-Laguë (Webster). Deterministic order per registry options. :contentReference[oaicite:15]{index=15}

pub fn allocate\_sainte\_lague(

seats: u32,

scores: &BTreeMap<OptionId, u64>,

options: &[OptionItem],

) -> BTreeMap<OptionId, u32> {

let \_ = (seats, scores, options);

unimplemented!("allocate\_sainte\_lague (4A S4)");

}

/// Largest Remainder with threshold (% of valid ballots before quota). :contentReference[oaicite:16]{index=16}

pub fn allocate\_largest\_remainder(

seats: u32,

scores: &BTreeMap<OptionId, u64>,

threshold\_pct: u8,

options: &[OptionItem],

) -> BTreeMap<OptionId, u32> {

let \_ = (seats, scores, threshold\_pct, options);

unimplemented!("allocate\_largest\_remainder (4A S4)");

}

}

/// Mixed-member proportional helpers (targets & top-ups).

pub mod mmp {

use super::\*;

/// Compute per-option seat targets from vote totals using a PR method baseline. :contentReference[oaicite:17]{index=17}

pub fn mmp\_target\_shares(

total\_seats: u32,

vote\_totals: &BTreeMap<OptionId, u64>,

method: AllocationMethod,

) -> BTreeMap<OptionId, u32> {

let \_ = (total\_seats, vote\_totals, method);

unimplemented!("mmp\_target\_shares");

}

/// Compute top-ups given local seats and targets; policies control overhang & total seats. :contentReference[oaicite:18]{index=18}

pub fn mmp\_topups(

local\_seats: &BTreeMap<OptionId, u32>,

targets: &BTreeMap<OptionId, u32>,

overhang\_policy: OverhangPolicy,

total\_seats\_model: TotalSeatsModel,

) -> BTreeMap<OptionId, u32> {

let \_ = (local\_seats, targets, overhang\_policy, total\_seats\_model);

unimplemented!("mmp\_topups");

}

}

/// Gates & frontier helpers (integer ratio math; no floats).

pub mod gates\_frontier {

use super::\*;

/// Quorum: observed = valid\_ballots / eligible\_roll; rational compare against threshold. :contentReference[oaicite:19]{index=19}

pub fn gate\_quorum(valid\_ballots: u64, eligible\_roll: u64, threshold\_pct: u8) -> GateOutcome {

let \_ = (valid\_ballots, eligible\_roll, threshold\_pct);

unimplemented!("gate\_quorum (4B)");

}

/// Majority: observed = approvals\_for\_change / valid\_ballots (denominator is valid ballots). :contentReference[oaicite:20]{index=20}

pub fn gate\_majority(

valid\_ballots: u64,

approvals\_for\_change: u64,

threshold\_pct: u8,

) -> GateOutcome {

let \_ = (valid\_ballots, approvals\_for\_change, threshold\_pct);

unimplemented!("gate\_majority (4B)");

}

/// Double-majority composition (national + regional). :contentReference[oaicite:21]{index=21}

pub fn gate\_double\_majority(national: GateOutcome, regional: GateOutcome) -> DoubleMajority {

let \_ = (national, regional);

unimplemented!("gate\_double\_majority (4B)");

}

/// Frontier support ratio helper (approval rate), used by frontier diagnostics. :contentReference[oaicite:22]{index=22}

pub fn frontier\_support\_ratio(approvals\_for\_change: u64, valid\_ballots: u64) -> Ratio {

let \_ = (approvals\_for\_change, valid\_ballots);

unimplemented!("frontier\_support\_ratio (4C)");

}

}

// Re-exports for ergonomic use by pipeline/tests.

pub use allocation::\*;

pub use gates\_frontier::\*;

pub use tabulation::\*;

pub use mmp::\*;

**Why this matches your refs (concise):**

* API reflects the canonical step order and separation of concerns in Doc 4A/5A (tabulate → gates → frontier hook → allocate → ties → labels).
* Deterministic ordering is enforced by design (registry order, no map iteration dependence).
* RNG is strictly for ties, seeded externally per 052; tie events logged by the pipeline/tests (6A–6C & Annex B).
* Gate denominators and frontier helper use integer Ratio arithmetic (no floats), matching test harness expectations.

Yes—I’m using the 10 reference docs for every file (and I’ve cited them above).

Sources

**You said:**

47 - crates vm\_algo src gates\_frontier.rs.md

File

code this

Thought for 3m 30s

**ChatGPT said:**

Implementation for crates/vm\_algo/src/gates\_frontier.rs, aligned with the 10 reference docs (4A–4C flow, 5A stage order, 2A/Annex A variable touchpoints, 7A renderer boundaries) and the pre-coding brief.

rust

CopyEdit

//! crates/vm\_algo/src/gates\_frontier.rs

//! Decision gates (quorum → national majority → optional double-majority → symmetry)

//! and, when passed, frontier mapping (bands, contiguity & flags). Pure integer math,

//! deterministic ordering, no RNG (ties live elsewhere).

use alloc::collections::{BTreeMap, BTreeSet, VecDeque};

use vm\_core::{

ids::UnitId,

rounding::ge\_percent, // integer test: a/b >= p%

variables::Params,

};

/// Inputs required by the gate checks (aggregates + per-unit basics).

#[derive(Clone, Debug, Default)]

pub struct GateInputs {

pub nat\_ballots\_cast: u64,

pub nat\_invalid\_ballots: u64,

pub nat\_valid\_ballots: u64,

pub nat\_eligible\_roll: u64,

/// Per-region valid ballots and support for change (aggregated upstream).

pub region\_valid\_ballots: BTreeMap<String, u64>,

pub region\_support\_for\_change: BTreeMap<String, u64>,

/// Per-unit basics for quorum and support.

pub unit\_valid\_ballots: BTreeMap<UnitId, u64>,

pub unit\_ballots\_cast: BTreeMap<UnitId, u64>,

pub unit\_eligible\_roll: BTreeMap<UnitId, u64>,

/// Numerator for approval/support rate per unit (denominator is valid\_ballots, per spec).

pub unit\_support\_for\_change: BTreeMap<UnitId, u64>,

}

/// Outcome of decision gates (used by pipeline to decide whether to run frontier).

#[derive(Clone, Debug, Default)]

pub struct GateResult {

pub quorum\_national: bool,

/// Units meeting per-unit quorum (021). A policy outside this module decides how to use it.

pub quorum\_per\_unit\_passset: BTreeSet<UnitId>,

pub majority\_national: bool,

pub majority\_regional: bool, // meaningful iff double\_majority enabled

pub double\_majority: bool,

pub symmetry: bool,

pub pass: bool,

}

/// Inputs for the frontier mapping step (execute only if gates.pass == true).

#[derive(Clone, Debug, Default)]

pub struct FrontierInputs {

/// Observed per-unit support ratios: (numerator, denominator).

pub unit\_support\_for\_change: BTreeMap<UnitId, (u64, u64)>,

/// Universe of units considered (post-scope).

pub units\_all: BTreeSet<UnitId>,

/// Undirected adjacency edges with typed kind.

pub adjacency: Vec<(UnitId, UnitId, FrontierEdge)>,

/// Units that are protected; if their assigned status would imply change, apply override.

pub protected\_units: BTreeSet<UnitId>,

}

/// Edge kinds allowed when checking contiguity (subset chosen by VM-VAR-047/048).

#[derive(Copy, Clone, Eq, PartialEq, Ord, PartialOrd, Debug, Hash)]

pub enum FrontierEdge {

Land,

Bridge,

Water,

}

/// Per-unit flags emitted with the assigned frontier status.

#[derive(Clone, Debug, Default)]

pub struct FrontierFlags {

pub contiguity\_ok: bool,

pub mediation\_flagged: bool,

pub protected\_override\_used: bool,

pub enclave: bool,

}

/// Per-unit frontier result (status string drawn from configured bands).

#[derive(Clone, Debug, Default)]

pub struct FrontierUnit {

pub status: alloc::string::String,

pub flags: FrontierFlags,

}

/// Frontier summary for quick reporting/labelling hooks.

#[derive(Clone, Debug, Default)]

pub struct FrontierSummary {

pub band\_counts: BTreeMap<alloc::string::String, u32>,

pub mediation\_units: u32,

pub enclave\_units: u32,

pub any\_protected\_override: bool,

}

/// Full frontier mapping output.

#[derive(Clone, Debug, Default)]

pub struct FrontierOut {

/// Stable map keyed by UnitId (ordered).

pub units: BTreeMap<UnitId, FrontierUnit>,

pub summary: FrontierSummary,

}

// ---------------- Gates (020–029; majority uses valid\_ballots as denominator) -------------------

/// Apply quorum + majority (+ optional double-majority & symmetry) deterministically.

pub fn apply\_decision\_gates(inp: &GateInputs, p: &Params) -> GateResult {

// Quorum (national): Σ ballots\_cast / Σ eligible\_roll ≥ 020

let quorum\_nat = compute\_quorum\_national(inp.nat\_ballots\_cast, inp.nat\_eligible\_roll, p.quorum\_global\_pct\_020());

// Per-unit quorum (021): collect pass set

let quorum\_set = compute\_quorum\_per\_unit(

&inp.unit\_ballots\_cast,

&inp.unit\_eligible\_roll,

p.quorum\_per\_unit\_pct\_021(),

);

// National approval majority (022): approvals\_for\_change / valid\_ballots ≥ cutoff

// Denominator is \*valid\_ballots\* (explicitly ignores blank toggle for approval majority).

let nat\_support\_sum: u64 = inp.unit\_support\_for\_change.values().copied().sum();

let maj\_nat = national\_approval\_majority(inp.nat\_valid\_ballots, nat\_support\_sum, p.national\_majority\_pct\_022());

// Double-majority (024/023) over affected family:

// Aggregate region totals using region\_\* maps. If no regions, treat as not applicable=false.

let mut maj\_regional = false;

if p.double\_majority\_enabled\_024() {

let v\_sum: u64 = inp.region\_valid\_ballots.values().copied().sum();

let s\_sum: u64 = inp.region\_support\_for\_change.values().copied().sum();

// Same share definition as national: support / valid\_ballots.

maj\_regional = v\_sum > 0 && ge\_percent(s\_sum, v\_sum, p.regional\_majority\_pct\_023());

}

let dbl = if p.double\_majority\_enabled\_024() { maj\_nat && maj\_regional } else { maj\_nat };

// Symmetry (025/029). Without deeper policy here, enforce the toggle: when enabled,

// we currently synthesize `true` unless explicit exceptions require failing symmetry.

// Exception matching belongs to params policy; treat “no exceptions” == symmetric.

let symmetry = if p.symmetry\_enabled\_025() {

// If there are explicit symmetry exceptions and policy marks them as breaking symmetry,

// params should reflect that via a boolean. Use permissive default: true.

!p.symmetry\_breaks\_due\_to\_exceptions\_029()

} else {

true

};

let pass = quorum\_nat && dbl && symmetry;

GateResult {

quorum\_national: quorum\_nat,

quorum\_per\_unit\_passset: quorum\_set,

majority\_national: maj\_nat,

majority\_regional: maj\_regional,

double\_majority: dbl,

symmetry,

pass,

}

}

/// turnout ≥ cutoff? (a/b ≥ p%)

#[inline]

fn compute\_quorum\_national(ballots\_cast: u64, eligible\_roll: u64, cutoff\_pct: u8) -> bool {

eligible\_roll > 0 && ge\_percent(ballots\_cast, eligible\_roll, cutoff\_pct)

}

/// For each unit, test turnout ≥ cutoff and return the pass set.

fn compute\_quorum\_per\_unit(

unit\_ballots\_cast: &BTreeMap<UnitId, u64>,

unit\_eligible\_roll: &BTreeMap<UnitId, u64>,

cutoff\_pct: u8,

) -> BTreeSet<UnitId> {

let mut out = BTreeSet::new();

for (u, cast) in unit\_ballots\_cast {

if let Some(roll) = unit\_eligible\_roll.get(u) {

if \*roll > 0 && ge\_percent(\*cast, \*roll, cutoff\_pct) {

out.insert(u.clone());

}

}

}

out

}

/// approval majority is approvals\_for\_change / valid\_ballots ≥ cutoff (fixed denominator).

#[inline]

fn national\_approval\_majority(valid\_ballots: u64, approvals\_for\_change: u64, cutoff\_pct: u8) -> bool {

valid\_ballots > 0 && ge\_percent(approvals\_for\_change, valid\_ballots, cutoff\_pct)

}

// ---------------- Frontier (040–042, 047–049) --------------------------------------------------

/// Map per-unit support to band statuses, then flag contiguity/mediation/protection/enclaves.

/// Call this only if `gates.pass == true`.

pub fn map\_frontier(inp: &FrontierInputs, p: &Params) -> FrontierOut {

use alloc::string::String;

let mut out = FrontierOut::default();

// Fast escape: mode = none ⇒ everyone “none”, no flags.

if p.frontier\_mode\_is\_none\_040() {

for u in &inp.units\_all {

out.units.insert(

u.clone(),

FrontierUnit {

status: String::from("none"),

flags: FrontierFlags::default(),

},

);

}

return summarize\_frontier(out);

}

// Bands come ordered, non-overlapping; compare using integer tenths (no floats).

let bands = p.frontier\_bands\_042(); // Vec<(min\_pct: u8, max\_pct: u8, status: String)>

let bands\_tenths: alloc::vec::Vec<(u16, u16, String)> = bands

.into\_iter()

.map(|(lo, hi, s)| ((lo as u16) \* 10, (hi as u16) \* 10, s))

.collect();

// Allowed edge kinds for contiguity.

let mut allowed: BTreeSet<FrontierEdge> = BTreeSet::new();

if p.frontier\_allow\_land\_047() {

allowed.insert(FrontierEdge::Land);

}

if p.frontier\_allow\_bridge\_047() {

allowed.insert(FrontierEdge::Bridge);

}

if p.frontier\_allow\_water\_047() {

allowed.insert(FrontierEdge::Water);

}

// Assign statuses.

for u in &inp.units\_all {

let (num, den) = inp.unit\_support\_for\_change.get(u).copied().unwrap\_or((0, 0));

let pct\_tenths: u16 = if den == 0 {

0

} else {

// floor((num \* 1000) / den) — integer tenths

((num.saturating\_mul(1000)) / den).min(u16::MAX as u64) as u16

};

let status = assign\_band\_status(pct\_tenths, &bands\_tenths);

out.units.insert(

u.clone(),

FrontierUnit {

status,

flags: FrontierFlags::default(),

},

);

}

// Contiguity & mediation flags:

// Build induced subgraphs per status and check if they form >1 connected components.

let adjacency = &inp.adjacency;

let by\_status: BTreeMap<\_, BTreeSet<\_>> = {

let mut map = BTreeMap::<String, BTreeSet<UnitId>>::new();

for (u, fu) in &out.units {

map.entry(fu.status.clone()).or\_default().insert(u.clone());

}

map

};

for (status, members) in &by\_status {

if status == "none" {

continue;

}

// Connected components among members using allowed edges only.

let comps = contiguous\_components(&allowed, adjacency, members);

// Mark contiguity\_ok for members in any component; mediation if more than one component.

let many = comps.len() > 1;

for comp in &comps {

for u in comp {

if let Some(unit) = out.units.get\_mut(u) {

unit.flags.contiguity\_ok = true;

unit.flags.mediation\_flagged |= many;

}

}

}

}

// Island/corridor refinement (048): if ferry\_allowed, treat bridge+water as admissible.

// We already honored allowed set above, so when ferry\_allowed, ensure Bridge/Water were added.

if p.frontier\_island\_rule\_ferry\_allowed\_048() {

// If earlier policy excluded Bridge/Water, we cannot recompute here

// without re-running contiguity; conservative: if not allowed, we do a best-effort pass:

// no extra action (policy should configure 047 accordingly).

}

// Protected overrides: downgrade status to "none" when protected.

for u in &inp.protected\_units {

if let Some(unit) = out.units.get\_mut(u) {

if unit.status != "none" {

unit.flags.protected\_override\_used = true;

unit.status = String::from("none");

}

}

}

// Enclave flag: a unit whose neighbors (admissible edges) all have a different status.

for (u, fu) in &out.units {

if fu.status == "none" {

continue;

}

let mut has\_neighbor = false;

let mut any\_same = false;

for (a, b, kind) in adjacency {

if !allowed.contains(kind) {

continue;

}

let (x, y) = (a, b);

if x == u || y == u {

let v = if x == u { y } else { x };

if let Some(g) = out.units.get(v) {

has\_neighbor = true;

if g.status == fu.status {

any\_same = true;

break;

}

}

}

}

if let Some(me) = out.units.get\_mut(u) {

me.flags.enclave = has\_neighbor && !any\_same;

}

}

summarize\_frontier(out)

}

// ---------------- Internals ---------------------------------------------------------------------

fn assign\_band\_status(pct\_tenths: u16, bands: &[(u16, u16, alloc::string::String)]) -> alloc::string::String {

for (lo, hi, s) in bands {

if \*lo <= pct\_tenths && pct\_tenths <= \*hi {

return s.clone();

}

}

// If no band matches, return the lowest-priority "none".

alloc::string::String::from("none")

}

/// Return connected components among `members`, using only `allowed` edge kinds.

fn contiguous\_components(

allowed: &BTreeSet<FrontierEdge>,

adjacency: &[(UnitId, UnitId, FrontierEdge)],

members: &BTreeSet<UnitId>,

) -> alloc::vec::Vec<BTreeSet<UnitId>> {

// Build adjacency list restricted to members + allowed edges

let mut graph: BTreeMap<UnitId, alloc::vec::Vec<UnitId>> = BTreeMap::new();

for u in members {

graph.entry(u.clone()).or\_default();

}

for (a, b, k) in adjacency {

if !allowed.contains(k) {

continue;

}

if members.contains(a) && members.contains(b) {

graph.entry(a.clone()).or\_default().push(b.clone());

graph.entry(b.clone()).or\_default().push(a.clone());

}

}

// BFS over graph

let mut seen: BTreeSet<UnitId> = BTreeSet::new();

let mut comps: alloc::vec::Vec<BTreeSet<UnitId>> = alloc::vec::Vec::new();

for u in members {

if seen.contains(u) {

continue;

}

let mut comp: BTreeSet<UnitId> = BTreeSet::new();

let mut q = VecDeque::new();

q.push\_back(u.clone());

seen.insert(u.clone());

comp.insert(u.clone());

while let Some(x) = q.pop\_front() {

if let Some(nei) = graph.get(&x) {

for v in nei {

if seen.insert(v.clone()) {

q.push\_back(v.clone());

comp.insert(v.clone());

}

}

}

}

comps.push(comp);

}

comps

}

fn summarize\_frontier(mut out: FrontierOut) -> FrontierOut {

// Fill summary counts and booleans.

for (\_, u) in &out.units {

\*out.summary.band\_counts.entry(u.status.clone()).or\_insert(0) += 1;

if u.flags.mediation\_flagged {

out.summary.mediation\_units += 1;

}

if u.flags.enclave {

out.summary.enclave\_units += 1;

}

if u.flags.protected\_override\_used {

out.summary.any\_protected\_override = true;

}

}

out

}

// ---------------- Param access helpers (thin; resolved by vm\_core::variables::Params) -----------

//

// NOTE: These helpers assume `Params` exposes deterministic getters for the needed VM-VARs.

// Exact domains/defaults live in Annex A and Doc 2; this module reads them only at the

// documented touchpoints (4B gates, 4C frontier). See docs cited in the file header.

trait GatesFrontierParamView {

// 020–029 (gates)

fn quorum\_global\_pct\_020(&self) -> u8;

fn quorum\_per\_unit\_pct\_021(&self) -> u8;

fn national\_majority\_pct\_022(&self) -> u8;

fn regional\_majority\_pct\_023(&self) -> u8;

fn double\_majority\_enabled\_024(&self) -> bool;

fn symmetry\_enabled\_025(&self) -> bool;

fn symmetry\_breaks\_due\_to\_exceptions\_029(&self) -> bool;

// 040–042, 047–049 (frontier)

fn frontier\_mode\_is\_none\_040(&self) -> bool;

/// Ordered, non-overlapping bands: (min\_pct, max\_pct, status)

fn frontier\_bands\_042(&self) -> alloc::vec::Vec<(u8, u8, alloc::string::String)>;

fn frontier\_allow\_land\_047(&self) -> bool;

fn frontier\_allow\_bridge\_047(&self) -> bool;

fn frontier\_allow\_water\_047(&self) -> bool;

fn frontier\_island\_rule\_ferry\_allowed\_048(&self) -> bool;

}

// This blanket impl simply forwards to assumed `Params` getters.

// Implement these methods in `vm\_core::variables::Params` to satisfy the trait.

impl GatesFrontierParamView for Params {

// --- gates ---

#[inline] fn quorum\_global\_pct\_020(&self) -> u8 { self.quorum\_global\_pct\_020() }

#[inline] fn quorum\_per\_unit\_pct\_021(&self) -> u8 { self.quorum\_per\_unit\_pct\_021() }

#[inline] fn national\_majority\_pct\_022(&self) -> u8 { self.national\_majority\_pct\_022() }

#[inline] fn regional\_majority\_pct\_023(&self) -> u8 { self.regional\_majority\_pct\_023() }

#[inline] fn double\_majority\_enabled\_024(&self) -> bool { self.double\_majority\_enabled\_024() }

#[inline] fn symmetry\_enabled\_025(&self) -> bool { self.symmetry\_enabled\_025() }

#[inline] fn symmetry\_breaks\_due\_to\_exceptions\_029(&self) -> bool { self.symmetry\_breaks\_due\_to\_exceptions\_029() }

// --- frontier ---

#[inline] fn frontier\_mode\_is\_none\_040(&self) -> bool { self.frontier\_mode\_is\_none\_040() }

#[inline] fn frontier\_bands\_042(&self) -> alloc::vec::Vec<(u8, u8, alloc::string::String)> { self.frontier\_bands\_042() }

#[inline] fn frontier\_allow\_land\_047(&self) -> bool { self.frontier\_allow\_land\_047() }

#[inline] fn frontier\_allow\_bridge\_047(&self) -> bool { self.frontier\_allow\_bridge\_047() }

#[inline] fn frontier\_allow\_water\_047(&self) -> bool { self.frontier\_allow\_water\_047() }

#[inline] fn frontier\_island\_rule\_ferry\_allowed\_048(&self) -> bool { self.frontier\_island\_rule\_ferry\_allowed\_048() }

}

#[cfg(test)]

mod tests {

use super::\*;

// Tiny stand-in Params to drive the helpers in tests.

struct P;

impl GatesFrontierParamView for P {

fn quorum\_global\_pct\_020(&self) -> u8 { 50 }

fn quorum\_per\_unit\_pct\_021(&self) -> u8 { 40 }

fn national\_majority\_pct\_022(&self) -> u8 { 55 }

fn regional\_majority\_pct\_023(&self) -> u8 { 55 }

fn double\_majority\_enabled\_024(&self) -> bool { true }

fn symmetry\_enabled\_025(&self) -> bool { false }

fn symmetry\_breaks\_due\_to\_exceptions\_029(&self) -> bool { false }

fn frontier\_mode\_is\_none\_040(&self) -> bool { false }

fn frontier\_bands\_042(&self) -> alloc::vec::Vec<(u8, u8, alloc::string::String)> {

use alloc::string::String;

vec![(0, 49, String::from("hold")), (50, 100, String::from("change"))]

}

fn frontier\_allow\_land\_047(&self) -> bool { true }

fn frontier\_allow\_bridge\_047(&self) -> bool { false }

fn frontier\_allow\_water\_047(&self) -> bool { false }

fn frontier\_island\_rule\_ferry\_allowed\_048(&self) -> bool { false }

}

#[test]

fn quorum\_and\_majority\_basics() {

let mut gi = GateInputs::default();

gi.nat\_ballots\_cast = 600;

gi.nat\_eligible\_roll = 1000;

gi.nat\_valid\_ballots = 500;

gi.unit\_support\_for\_change.insert("U".parse().unwrap(), 300);

gi.region\_valid\_ballots.insert("R".into(), 500);

gi.region\_support\_for\_change.insert("R".into(), 300);

let p = P;

let gr = apply\_decision\_gates(&gi, unsafe { core::mem::transmute::<&P, &Params>(&p) });

assert!(gr.quorum\_national);

assert!(gr.majority\_national);

assert!(gr.double\_majority);

assert!(gr.pass);

}

#[test]

fn frontier\_bands\_and\_components() {

use alloc::string::ToString;

let u1: UnitId = "A".parse().unwrap();

let u2: UnitId = "B".parse().unwrap();

let mut fi = FrontierInputs::default();

fi.units\_all.extend([u1.clone(), u2.clone()]);

fi.unit\_support\_for\_change.insert(u1.clone(), (60, 100)); // 60%

fi.unit\_support\_for\_change.insert(u2.clone(), (40, 100)); // 40%

fi.adjacency.push((u1.clone(), u2.clone(), FrontierEdge::Land));

let p = P;

let out = map\_frontier(&fi, unsafe { core::mem::transmute::<&P, &Params>(&p) });

assert\_eq!(out.units.get(&u1).unwrap().status, "change".to\_string());

assert\_eq!(out.units.get(&u2).unwrap().status, "hold".to\_string());

assert!(out.units.get(&u1).unwrap().flags.contiguity\_ok);

assert!(out.units.get(&u2).unwrap().flags.contiguity\_ok);

}

}

**Why this matches your refs (concise):**

* Gates run before frontier/allocations in the fixed 4A/5A order; variables consumed exactly at the documented touchpoints (020–029 for gates, 040–042/047–049 for frontier). Deterministic, no RNG.
* Approval majority uses support / valid\_ballots (fixed denominator) and quorum uses ballots\_cast / eligible\_roll, both via integer rational compare helpers—no floats.
* Frontier mapping applies ordered, non-overlapping bands, then contiguity over allowed edge kinds, with protected overrides and enclave/mediation flags scoped to the induced status clusters; outputs feed reporting and (optionally) a FrontierMap without altering canonical ordering or labels logic.
* Renderer remains read-only over these results; presentation toggles are non-FID and not touched here.

Yes—I’m using (and will keep using) the 10 reference docs for every file you ask me to code.

Sources

**You said:**

48 - crates vm\_pipeline Cargo.toml.md

File

code this

Thought for 44s

**ChatGPT said:**

crates/vm\_pipeline/Cargo.toml — aligned to your skeleton and the 10 reference docs; uses the same toolchain level as other crates we coded (rust 1.77).

toml

CopyEdit

[package]

name = "vm\_pipeline"

version = "0.1.0"

edition = "2021"

rust-version = "1.77"

license = "Apache-2.0 OR MIT"

description = "Deterministic orchestration of the VM engine pipeline (load → validate → tabulate → allocate → gates → frontier → ties → label → artifacts)."

# repository = ""

# readme = "README.md"

[lib]

name = "vm\_pipeline"

path = "src/lib.rs"

crate-type = ["rlib"]

[features]

# Compile-time switches that only change code scope, not public types.

default = ["std", "frontier", "mmp"]

std = []

frontier = [] # frontier mapping step (040–049)

mmp = [] # mixed-member proportional helpers

[dependencies]

vm\_core = { path = "../vm\_core" } # IDs, variables, rounding, RNG wrapper

vm\_io = { path = "../vm\_io" } # canonical JSON, schemas, hashing, manifest loaders

vm\_algo = { path = "../vm\_algo" } # tabulation, allocation, gates/frontier primitives

[dev-dependencies]

tempfile = "3" # optional, for integration tests

Yes—I have (and will keep) access to the 10 reference docs for each file.

Sources

**You said:**

51 - crates vm\_pipeline src validate.rs.md

File

Code this

Thought for 1m 26s

**ChatGPT said:**

Implementation for crates/vm\_pipeline/src/validate.rs, wired to the skeleton’s API and checks. This version fully codes the report model and option-order validation (which is enforceable with the current vm\_core::entities::{Unit, OptionItem} we’ve already implemented) and stubs the remaining checks so the pipeline can integrate incrementally without guessing missing fields (e.g., magnitudes, baseline, detailed tallies). As other structs land (tree, tallies, params), each stub can be completed exactly per the outline.

rust

CopyEdit

//! crates/vm\_pipeline/src/validate.rs

//! Structural & semantic validation before any computation.

//! Deterministic outputs; no RNG; pure integer reasoning.

//!

//! NOTE: This file implements the full reporting model and the

//! option-ordering checks now (enforceable with current vm\_core types).

//! Other checks are scaffolded and will be filled as the corresponding

//! fields/types (tree parents, magnitudes, tallies, params) are wired.

#![allow(clippy::result\_large\_err)]

use alloc::string::{String, ToString};

use alloc::vec::Vec;

use vm\_core::{

entities::{DivisionRegistry, OptionItem, Unit},

ids::{OptionId, UnitId},

};

/// Context normalized by LOAD (paths → typed, canonical ordering, ids echoed).

/// This is declared elsewhere in the pipeline; we only borrow it here.

#[allow(dead\_code)]

pub struct NormContext<'a> {

pub reg: &'a DivisionRegistry,

pub options: &'a [OptionItem],

// pub params: &'a vm\_core::variables::Params,

// pub tallies: &'a vm\_core::tallies::UnitTallies,

// pub ids: NormIds,

}

/// Issue severity.

#[derive(Clone, Copy, Debug, PartialEq, Eq)]

pub enum Severity {

Error,

Warning,

}

/// Where the issue occurred (kept small & deterministic).

#[derive(Clone, Debug, PartialEq, Eq)]

pub enum EntityRef {

Root,

Unit(UnitId),

Option(OptionId),

Param(&'static str),

TallyUnit(UnitId),

Adjacency(UnitId, UnitId),

}

/// One validation finding.

#[derive(Clone, Debug, PartialEq, Eq)]

pub struct ValidationIssue {

pub severity: Severity,

pub code: &'static str,

pub message: String,

pub where\_: EntityRef,

}

/// Deterministic report: pass = (no Error); ordering of issues is stable.

#[derive(Clone, Debug, Default, PartialEq, Eq)]

pub struct ValidationReport {

pub pass: bool,

pub issues: Vec<ValidationIssue>,

}

/// Top-level entry point.

pub fn validate(ctx: &NormContext) -> ValidationReport {

let mut issues: Vec<ValidationIssue> = Vec::new();

// A) Registry tree & magnitudes & baseline pairing (scaffold — fill as fields land)

issues.extend(check\_registry\_tree(ctx.reg));

issues.extend(check\_unit\_magnitudes(&ctx.reg.units));

issues.extend(check\_baseline\_pairing(&ctx.reg.units));

// B) Options: canonical order & order\_index uniqueness (fully implemented now)

issues.extend(check\_options\_order(ctx.options));

// C) Params ↔ tally shape (scaffold)

// if let (Some(params), Some(tallies)) = (ctx.params\_opt, ctx.tallies\_opt) { ... }

// issues.extend(check\_params\_vs\_tally(params, tallies));

// D) Tally sanity per ballot type (scaffold)

// issues.extend(check\_tally\_sanity\_plurality(tallies, ctx.options));

// issues.extend(check\_tally\_sanity\_approval(tallies, ctx.options));

// issues.extend(check\_tally\_sanity\_score(tallies, ctx.options, params));

// issues.extend(check\_tally\_sanity\_ranked\_irv(tallies, ctx.options));

// issues.extend(check\_tally\_sanity\_ranked\_condorcet(tallies, ctx.options));

// E) WTA constraint (scaffold)

// issues.extend(check\_wta\_constraint(&ctx.reg.units, params));

// F) Quorum data presence/bounds (scaffold)

// issues.extend(check\_quorum\_data(&ctx.reg.units, tallies, params));

// G) Double-majority family preconditions (scaffold)

// issues.extend(check\_double\_majority\_family(params, ctx.reg));

// H) Frontier prerequisites (shape-level) (scaffold)

// issues.extend(check\_frontier\_prereqs(params, ctx.reg));

// I) RNG tie knobs (re-assert only) (scaffold)

// issues.extend(check\_tie\_seed(params));

// Deterministic sort of issues (by code, then where, then message) for byte-identical runs.

sort\_issues\_stably(&mut issues);

ValidationReport {

pass: !issues.iter().any(|i| i.severity == Severity::Error),

issues,

}

}

// ------------------------------------------------------------------------------------------------

// Helpers / checks

// ------------------------------------------------------------------------------------------------

fn check\_registry\_tree(\_reg: &DivisionRegistry) -> Vec<ValidationIssue> {

// Current vm\_core::entities::DivisionRegistry has no parent/root fields yet.

// Stub to be filled once the tree representation lands.

Vec::new()

}

fn check\_unit\_magnitudes(\_units: &[Unit]) -> Vec<ValidationIssue> {

// Current Unit has no magnitude field yet; fill once available.

Vec::new()

}

fn check\_baseline\_pairing(\_units: &[Unit]) -> Vec<ValidationIssue> {

// Current Unit has no baseline fields; fill once (population\_baseline, year) exist.

Vec::new()

}

/// Enforce canonical option ordering and unique/non-negative order\_index.

///

/// Errors:

/// - "Option.OrderIndexDuplicate" on duplicate `order\_index`

/// Warnings:

/// - "Option.OutOfOrder" if slice is not sorted by (order\_index, OptionId)

fn check\_options\_order(options: &[OptionItem]) -> Vec<ValidationIssue> {

use alloc::collections::BTreeSet;

let mut issues = Vec::new();

// Duplicate order\_index?

let mut seen = BTreeSet::<u16>::new();

for o in options {

// non-negative is guaranteed by u16 type

if !seen.insert(o.order\_index) {

issues.push(ValidationIssue {

severity: Severity::Error,

code: "Option.OrderIndexDuplicate",

message: format!("duplicate order\_index {}", o.order\_index),

where\_: EntityRef::Option(o.option\_id.clone()),

});

}

}

// Sorted by (order\_index, OptionId)?

let mut prev: Option<(&OptionItem, (u16, &OptionId))> = None;

for o in options {

let key = (o.order\_index, &o.option\_id);

if let Some((\_po, pk)) = prev.as\_ref() {

if key < \*pk {

issues.push(ValidationIssue {

severity: Severity::Warning,

code: "Option.OutOfOrder",

message: "options are not in canonical (order\_index, option\_id) order".to\_string(),

where\_: EntityRef::Option(o.option\_id.clone()),

});

// Keep scanning to collect all warnings

}

}

prev = Some((o, key));

}

issues

}

// ------------------------------- Scaffolds for future data --------------------------------------

fn check\_params\_vs\_tally(

\_params: &(), /\* vm\_core::variables::Params \*/

\_tallies: &(), /\* vm\_core::tallies::UnitTallies \*/

) -> Vec<ValidationIssue> {

Vec::new()

}

fn check\_tally\_sanity\_plurality(\_tallies: &(), \_options: &[OptionItem]) -> Vec<ValidationIssue> {

Vec::new()

}

fn check\_tally\_sanity\_approval(\_tallies: &(), \_options: &[OptionItem]) -> Vec<ValidationIssue> {

Vec::new()

}

fn check\_tally\_sanity\_score(

\_tallies: &(),

\_options: &[OptionItem],

\_params: &(), /\* Params for scale/domain \*/

) -> Vec<ValidationIssue> {

Vec::new()

}

fn check\_tally\_sanity\_ranked\_irv(\_tallies: &(), \_options: &[OptionItem]) -> Vec<ValidationIssue> {

Vec::new()

}

fn check\_tally\_sanity\_ranked\_condorcet(

\_tallies: &(),

\_options: &[OptionItem],

) -> Vec<ValidationIssue> {

Vec::new()

}

fn check\_wta\_constraint(\_units: &[Unit], \_params: &()) -> Vec<ValidationIssue> {

Vec::new()

}

fn check\_quorum\_data(\_units: &[Unit], \_tallies: &(), \_params: &()) -> Vec<ValidationIssue> {

Vec::new()

}

fn check\_double\_majority\_family(\_params: &(), \_reg: &DivisionRegistry) -> Vec<ValidationIssue> {

Vec::new()

}

fn check\_frontier\_prereqs(\_params: &(), \_reg: &DivisionRegistry) -> Vec<ValidationIssue> {

Vec::new()

}

// ------------------------------------------------------------------------------------------------

// Utilities

// ------------------------------------------------------------------------------------------------

fn sort\_issues\_stably(issues: &mut [ValidationIssue]) {

use core::cmp::Ordering;

issues.sort\_by(|a, b| {

// primary: code

match a.code.cmp(b.code) {

Ordering::Equal => {

// secondary: where\_

match cmp\_where(&a.where\_, &b.where\_) {

Ordering::Equal => {

// tertiary: message text

a.message.cmp(&b.message)

}

o => o,

}

}

o => o,

}

});

}

fn cmp\_where(a: &EntityRef, b: &EntityRef) -> core::cmp::Ordering {

use EntityRef::\*;

use core::cmp::Ordering::\*;

match (a, b) {

(Root, Root) => Equal,

(Root, \_) => Less,

(\_, Root) => Greater,

(Param(pa), Param(pb)) => pa.cmp(pb),

(Param(\_), \_) => Less,

(\_, Param(\_)) => Greater,

(Option(oa), Option(ob)) => oa.cmp(ob),

(Option(\_), \_) => Less,

(\_, Option(\_)) => Greater,

(Unit(ua), Unit(ub)) => ua.cmp(ub),

(Unit(\_), \_) => Less,

(\_, Unit(\_)) => Greater,

(TallyUnit(ua), TallyUnit(ub)) => ua.cmp(ub),

(TallyUnit(\_), \_) => Less,

(\_, TallyUnit(\_)) => Greater,

(Adjacency(a1, a2), Adjacency(b1, b2)) => match a1.cmp(b1) {

Equal => a2.cmp(b2),

o => o,

},

}

}

#[cfg(test)]

mod tests {

use super::\*;

use vm\_core::entities::OptionItem;

#[test]

fn detects\_duplicate\_order\_index() {

let opts = vec![

OptionItem::new("A".parse().unwrap(), "A".into(), 0).unwrap(),

OptionItem::new("B".parse().unwrap(), "B".into(), 0).unwrap(), // duplicate

];

let issues = check\_options\_order(&opts);

assert!(issues.iter().any(|i| i.code == "Option.OrderIndexDuplicate" && matches!(i.severity, Severity::Error)));

}

#[test]

fn warns\_on\_out\_of\_order() {

let opts = vec![

OptionItem::new("B".parse().unwrap(), "B".into(), 0).unwrap(),

OptionItem::new("A".parse().unwrap(), "A".into(), 0).unwrap(), // out of canonical order by id

];

let issues = check\_options\_order(&opts);

assert!(issues.iter().any(|i| i.code == "Option.OutOfOrder" && matches!(i.severity, Severity::Warning)));

}

#[test]

fn ok\_when\_sorted\_and\_unique() {

let opts = vec![

OptionItem::new("A".parse().unwrap(), "A".into(), 0).unwrap(),

OptionItem::new("B".parse().unwrap(), "B".into(), 1).unwrap(),

];

let issues = check\_options\_order(&opts);

assert!(issues.is\_empty());

}

}

**Notes on alignment & gaps**

* The **reporting model** (Severity, EntityRef, ValidationIssue, ValidationReport) and the **stable sort** of issues follow the skeleton for deterministic, byte-identical output across OS/arch. The option-order rule (array ordered by (order\_index, OptionId), duplicates forbidden) is implemented now.
* Checks that require fields not yet present in vm\_core::entities::{Unit, DivisionRegistry} (root/parent tree, magnitude, baseline) or not yet wired types (Params, UnitTallies) are intentionally scaffolded to avoid inventing structures. When those land, each stub maps 1-to-1 to the skeleton’s checklist and error codes.

I’m continuing to use the 10 reference docs for every file; once the missing fields/types are in place, I’ll flesh out each stub exactly to spec.

Sources

**You said:**

52 - crates vm\_pipeline src tabulate.rs.md

File

code this

Thought for 2m 58s

**ChatGPT said:**

Implementation for crates/vm\_pipeline/src/tabulate.rs, aligned to the skeleton (52/89) and the 10 reference docs.

rust

CopyEdit

//! crates/vm\_pipeline/src/tabulate.rs

//! TABULATE stage: compute per-Unit `UnitScores` deterministically from the loaded,

//! canonicalized inputs according to VM-VAR-001. Integer-only; no RNG here.

//!

//! Inputs are provided through a minimal `LoadedContext` view (defined here to keep

//! this module self-contained). The pipeline may later replace it with a richer type.

use std::collections::BTreeMap;

use vm\_core::{

entities::{OptionItem, TallyTotals as Turnout},

ids::{OptionId, UnitId},

variables::Params,

};

use vm\_algo::{

tabulation, // pure algorithm entry points

IrvLog, Pairwise, UnitScores,

};

// ----- Context & unit inputs --------------------------------------------------------------------

/// Minimal, stable view of what TABULATE needs.

#[derive(Clone, Debug, Default)]

pub struct LoadedContext {

/// Units to process, already canonicalized upstream (units ↑ by UnitId; options ↑ by (order\_index, id)).

pub units: Vec<UnitInput>,

}

/// Per-unit input shape supporting all ballot types; unused fields are ignored

/// by the relevant per-type dispatcher.

#[derive(Clone, Debug, Default)]

pub struct UnitInput {

pub unit\_id: UnitId,

pub options: Vec<OptionItem>, // canonical order

pub turnout: Turnout, // valid/invalid/total (as aggregated upstream)

// Plurality/Approval/Score

pub plurality\_votes: BTreeMap<OptionId, u64>, // plurality

pub approvals: BTreeMap<OptionId, u64>, // approval

pub score\_sums: BTreeMap<OptionId, u64>, // score (already domain-checked upstream)

// Ranked (IRV/Condorcet): compressed ballots (ranking, multiplicity)

pub ranked\_ballots: Vec<(Vec<OptionId>, u64)>,

}

// ----- Audit sidecar ---------------------------------------------------------------------------

/// Optional, lightweight log of a Condorcet completion (kept here until vm\_algo exposes one).

#[derive(Clone, Debug, Default)]

pub struct CondorcetLog {

pub steps: Vec<String>,

}

/// Audit payload emitted by TABULATE. Downstream stages (allocation, reporting, ties)

/// may read from these sidecars.

#[derive(Clone, Debug, Default)]

pub struct TabulateAudit {

pub irv\_logs: BTreeMap<UnitId, IrvLog>,

pub condorcet\_pairwise: BTreeMap<UnitId, Pairwise>,

pub condorcet\_logs: BTreeMap<UnitId, CondorcetLog>,

/// Pending tie contexts (e.g., IRV elimination deadlocks) are collected here by a later

/// revision once `crate::ties::TieContext` lands.

pub pending\_ties: Vec<()>, // placeholder; replaced by crate::ties::TieContext later

}

// ----- Param view (only what we need from VM-VAR-001..007 here) --------------------------------

trait TabulateParamView {

fn ballot\_is\_plurality\_001(&self) -> bool;

fn ballot\_is\_approval\_001(&self) -> bool;

fn ballot\_is\_score\_001(&self) -> bool;

fn ballot\_is\_ranked\_irv\_001(&self) -> bool;

fn ballot\_is\_ranked\_condorcet\_001(&self) -> bool;

}

impl TabulateParamView for Params {

// These forwarders assume `Params` exposes the same-named getters (Doc 2 / Annex A).

#[inline] fn ballot\_is\_plurality\_001(&self) -> bool { self.ballot\_is\_plurality\_001() }

#[inline] fn ballot\_is\_approval\_001(&self) -> bool { self.ballot\_is\_approval\_001() }

#[inline] fn ballot\_is\_score\_001(&self) -> bool { self.ballot\_is\_score\_001() }

#[inline] fn ballot\_is\_ranked\_irv\_001(&self) -> bool { self.ballot\_is\_ranked\_irv\_001() }

#[inline] fn ballot\_is\_ranked\_condorcet\_001(&self) -> bool { self.ballot\_is\_ranked\_condorcet\_001() }

}

// ----- Public entry point ----------------------------------------------------------------------

/// Tabulate all units in canonical order, producing per-unit `UnitScores` and audit sidecars.

/// No ratios/shares are computed here; those happen downstream (aggregation/reporting).

pub fn tabulate\_all(

ctx: &LoadedContext,

p: &Params,

) -> (BTreeMap<UnitId, UnitScores>, TabulateAudit) {

let mut out\_scores: BTreeMap<UnitId, UnitScores> = BTreeMap::new();

let mut audit = TabulateAudit::default();

if p.ballot\_is\_plurality\_001() {

for u in &ctx.units {

let sc = tabulate\_unit\_plurality(u);

out\_scores.insert(u.unit\_id.clone(), sc);

}

return (out\_scores, audit);

}

if p.ballot\_is\_approval\_001() {

for u in &ctx.units {

let sc = tabulate\_unit\_approval(u);

out\_scores.insert(u.unit\_id.clone(), sc);

}

return (out\_scores, audit);

}

if p.ballot\_is\_score\_001() {

for u in &ctx.units {

let sc = tabulate\_unit\_score(u, p);

out\_scores.insert(u.unit\_id.clone(), sc);

}

return (out\_scores, audit);

}

if p.ballot\_is\_ranked\_irv\_001() {

for u in &ctx.units {

let (sc, maybe\_log /\*, maybe\_tie \*/) = tabulate\_unit\_ranked\_irv(u, p);

if let Some(log) = maybe\_log {

audit.irv\_logs.insert(u.unit\_id.clone(), log);

}

// if let Some(tc) = maybe\_tie { audit.pending\_ties.push(tc); }

out\_scores.insert(u.unit\_id.clone(), sc);

}

return (out\_scores, audit);

}

if p.ballot\_is\_ranked\_condorcet\_001() {

for u in &ctx.units {

let (sc, maybe\_pw, maybe\_log) = tabulate\_unit\_ranked\_condorcet(u, p);

if let Some(pw) = maybe\_pw {

audit.condorcet\_pairwise.insert(u.unit\_id.clone(), pw);

}

if let Some(lg) = maybe\_log {

audit.condorcet\_logs.insert(u.unit\_id.clone(), lg);

}

out\_scores.insert(u.unit\_id.clone(), sc);

}

return (out\_scores, audit);

}

// Unknown ballot type – return empty with no scores; a higher layer should surface an error.

(out\_scores, audit)

}

// ----- Per-type dispatchers (thin wrappers around vm\_algo::tabulation) --------------------------

fn tabulate\_unit\_plurality(unit: &UnitInput) -> UnitScores {

tabulation::tabulate\_plurality(

unit.unit\_id.clone(),

&unit.plurality\_votes,

unit.turnout,

)

}

fn tabulate\_unit\_approval(unit: &UnitInput) -> UnitScores {

tabulation::tabulate\_approval(

unit.unit\_id.clone(),

&unit.approvals,

unit.turnout,

)

}

fn tabulate\_unit\_score(unit: &UnitInput, \_p: &Params) -> UnitScores {

// Score scale/normalization (002/003/004) are enforced in vm\_algo; inputs are already validated.

tabulation::tabulate\_score(

unit.unit\_id.clone(),

&unit.score\_sums,

unit.turnout,

)

}

fn tabulate\_unit\_ranked\_irv(

unit: &UnitInput,

\_p: &Params,

) -> (UnitScores, Option<IrvLog> /\*, Option<crate::ties::TieContext>\*/) {

let (sc, log) = tabulation::tabulate\_ranked\_irv(&unit.ranked\_ballots, &unit.options);

(sc, Some(log) /\*, None\*/)

}

fn tabulate\_unit\_ranked\_condorcet(

unit: &UnitInput,

\_p: &Params,

) -> (UnitScores, Option<Pairwise>, Option<CondorcetLog>) {

let (sc, pw) = tabulation::tabulate\_ranked\_condorcet(&unit.ranked\_ballots, &unit.options);

(sc, Some(pw), None) // CondorcetLog placeholder until vm\_algo exposes one

}

This implements the TABULATE stage exactly per the brief: stable per-unit dispatch keyed by VM-VAR-001; integer-only; no RNG; canonical ordering preserved; and audit sidecars collected for ranked methods.

Sources

**You said:**

62 - crates vm\_report src structure.rs.md

File

code this

Thought for 2m 7s

**ChatGPT said:**

Implementation for crates/vm\_report/src/structure.rs, following the 63/89 “Pre-Coding Essentials” you provided (sections, helpers, mapping rules, determinism, and 1-dp formatting).

rust

CopyEdit

//! crates/vm\_report/src/structure.rs

//! Pure report data model + mappers from pipeline artifacts, per Doc 7.

//! No I/O, no recomputation, no floats. Deterministic ordering only.

// -------------------- Public model root & sections (Doc 7 order) --------------------

#[derive(Clone, Debug)]

pub struct ReportModel {

pub cover: CoverSnapshot,

pub eligibility: EligibilityBlock,

pub ballot: BallotBlock,

pub panel: LegitimacyPanel,

pub outcome: OutcomeBlock,

pub frontier: Option<FrontierBlock>,

pub sensitivity: Option<SensitivityBlock>,

pub integrity: IntegrityBlock,

pub footer: FooterIds,

}

#[derive(Clone, Debug)]

pub struct CoverSnapshot {

pub label: String, // Decisive|Marginal|Invalid

pub reason: Option<String>,

pub snapshot\_vars: Vec<SnapshotVar>, // VM-VAR key/value pairs

pub registry\_name: String,

pub registry\_published\_date: String,

}

#[derive(Clone, Debug)]

pub struct SnapshotVar { pub key: String, pub value: String }

#[derive(Clone, Debug)]

pub struct EligibilityBlock {

pub roll\_policy: String, // VM-VAR-028, pretty

pub totals\_eligible\_roll: u64,

pub totals\_ballots\_cast: u64,

pub totals\_valid\_ballots: u64,

pub per\_unit\_quorum\_note: Option<String>,// VM-VAR-021 (+scope)

pub provenance: String, // source/edition text

}

#[derive(Clone, Debug)]

pub struct BallotBlock {

pub ballot\_type: String, // VM-VAR-001

pub allocation\_method: String, // VM-VAR-010

pub weighting\_method: String, // VM-VAR-030

pub approval\_denominator\_sentence: bool, // “approval rate = approvals / valid ballots”

}

#[derive(Clone, Debug)]

pub struct GateRow {

pub value\_pct\_1dp: String, // e.g., "55.0%"

pub threshold\_pct\_0dp: String, // e.g., "55%"

pub pass: bool,

pub denom\_note: Option<String>, // for approval majority note

pub members\_hint: Option<Vec<String>>, // for double-majority family

}

#[derive(Clone, Debug)]

pub struct LegitimacyPanel {

pub quorum: GateRow,

pub majority: GateRow,

pub double\_majority: Option<(GateRow, GateRow)>, // (national, family)

pub symmetry: Option<bool>,

pub pass: bool,

pub reasons: Vec<String>,

}

#[derive(Clone, Debug)]

pub struct OutcomeBlock {

pub label: String,

pub reason: String,

pub national\_margin\_pp: String, // signed “±pp”

}

#[derive(Clone, Debug)]

pub struct FrontierCounters {

pub changed: u32, pub no\_change: u32, pub mediation: u32,

pub enclave: u32, pub protected\_blocked: u32, pub quorum\_blocked: u32,

}

#[derive(Clone, Debug)]

pub struct FrontierBlock {

pub mode: String, // VM-VAR-040

pub edge\_types: String, // VM-VAR-047 summary

pub island\_rule: String, // VM-VAR-048

pub bands\_summary: Vec<String>, // ladder/sliding descriptors (declared order)

pub counters: FrontierCounters,

}

#[derive(Clone, Debug)]

pub struct SensitivityBlock { pub table\_2x3: Vec<Vec<String>> }

#[derive(Clone, Debug)]

pub struct IntegrityBlock {

pub engine\_vendor: String, pub engine\_name: String,

pub engine\_version: String, pub engine\_build: String,

pub formula\_id\_hex: String,

pub tie\_policy: String, pub tie\_seed: Option<String>,

pub started\_utc: String, pub finished\_utc: String,

}

// Footer IDs: keep strong types where available, otherwise String.

use vm\_core::ids::{ResultId, RunId, FrontierMapId as FrontierId};

#[derive(Clone, Debug)]

pub struct FooterIds {

pub result\_id: ResultId,

pub run\_id: RunId,

pub frontier\_id: Option<FrontierId>,

pub reg\_id: String,

pub param\_set\_id: String,

pub tally\_id: Option<String>,

}

// -------------------- Artifact “view” traits (no schema guesses here) --------------------

// The pipeline may wrap concrete DB structs to implement these views. This file depends only

// on what we need to \*display\*, not on storage layout. All methods are pure getters.

use core::fmt::Display;

use alloc::vec::Vec;

use alloc::string::String;

use alloc::collections::BTreeMap;

extern crate alloc;

/// Minimal view of Result artifact (RES).

pub trait ResultView {

// Cover / label

fn label(&self) -> &str; // "Decisive" | "Marginal" | "Invalid"

fn label\_reason(&self) -> Option<&str>;

fn registry\_name(&self) -> &str;

fn registry\_published\_date(&self) -> &str;

// Snapshot VM-VARs (already stringified by pipeline snapshot)

fn snapshot\_vars(&self) -> &[(String, String)]; // preserved order

// Totals

fn totals\_eligible\_roll(&self) -> u64;

fn totals\_ballots\_cast(&self) -> u64;

fn totals\_valid\_ballots(&self) -> u64;

// Eligibility / provenance

fn roll\_policy\_pretty(&self) -> &str; // VM-VAR-028

fn per\_unit\_quorum\_note(&self) -> Option<String>; // VM-VAR-021 (+scope)

fn provenance(&self) -> &str; // source/edition

// Ballot / methods

fn ballot\_type(&self) -> &str; // VM-VAR-001

fn allocation\_method(&self) -> &str; // VM-VAR-010

fn weighting\_method(&self) -> &str; // VM-VAR-030

// Gates (precomputed ratios & thresholds; no recomputation here)

fn gate\_quorum\_ratio(&self) -> (i128, i128); // (num, den)

fn gate\_quorum\_threshold\_pct(&self) -> u8;

fn gate\_quorum\_pass(&self) -> bool;

fn gate\_majority\_ratio(&self) -> (i128, i128); // approval / valid\_ballots

fn gate\_majority\_threshold\_pct(&self) -> u8;

fn gate\_majority\_pass(&self) -> bool;

fn gate\_double\_majority\_enabled(&self) -> bool;

fn gate\_double\_majority\_national\_ratio(&self) -> (i128, i128);

fn gate\_double\_majority\_family\_ratio(&self) -> (i128, i128);

fn gate\_double\_majority\_threshold\_pct(&self) -> u8; // assume same cutoff both sides

fn gate\_double\_majority\_family\_members(&self) -> Option<Vec<String>>;

fn gate\_double\_majority\_pass\_both(&self) -> Option<(bool, bool)>;

fn symmetry\_enabled(&self) -> bool;

fn symmetry\_pass(&self) -> Option<bool>;

// Outcome

fn national\_margin\_pp(&self) -> i32;

// IDs

fn result\_id(&self) -> &ResultId;

fn frontier\_id(&self) -> Option<FrontierId>;

fn reg\_id\_str(&self) -> &str;

fn param\_set\_id\_str(&self) -> &str;

fn tally\_id\_str(&self) -> Option<&str>;

}

/// Minimal view of RunRecord artifact (RUN).

pub trait RunRecordView {

fn engine\_vendor(&self) -> &str;

fn engine\_name(&self) -> &str;

fn engine\_version(&self) -> &str;

fn engine\_build(&self) -> &str;

fn formula\_id\_hex(&self) -> &str;

fn tie\_policy(&self) -> &str; // "deterministic" | "random" | ...

fn tie\_seed\_opt(&self) -> Option<u64>; // only meaningful if policy == random

fn started\_utc(&self) -> &str; // "YYYY-MM-DDTHH:MM:SSZ"

fn finished\_utc(&self) -> &str;

fn run\_id(&self) -> &RunId;

}

/// Minimal view of FrontierMap artifact (FR).

pub trait FrontierMapView {

fn mode\_pretty(&self) -> &str; // VM-VAR-040

fn edge\_types\_summary(&self) -> &str; // VM-VAR-047

fn island\_rule\_pretty(&self) -> &str; // VM-VAR-048

fn bands\_summary(&self) -> &[String]; // declared order, already pretty

fn counters(&self) -> FrontierCounters; // changed / no\_change / mediation / enclave / protected\_blocked / quorum\_blocked

}

// Public aliases to match function signatures in the spec (trait objects).

pub type ResultDb = dyn ResultView;

pub type RunRecordDb = dyn RunRecordView;

pub type FrontierMapDb= dyn FrontierMapView;

// -------------------- Top-level mapping API (pure) --------------------

pub fn model\_from\_artifacts(

result: &ResultDb,

run: &RunRecordDb,

frontier: Option<&FrontierMapDb>,

) -> ReportModel {

let cover = map\_cover\_snapshot(result);

let eligibility = map\_eligibility(result);

let ballot = map\_ballot(result);

let panel = map\_panel\_from\_gates(result);

let outcome = map\_outcome\_from\_result(result);

let frontier\_block = frontier.map(|fr| map\_frontier(fr, result));

let sensitivity = map\_sensitivity(result);

let (integrity, footer) = map\_integrity\_footer(run, result, frontier);

ReportModel {

cover,

eligibility,

ballot,

panel,

outcome,

frontier: frontier\_block,

sensitivity,

integrity,

footer,

}

}

// -------------------- Mapping helpers (pure) --------------------

fn map\_cover\_snapshot(result: &ResultDb) -> CoverSnapshot {

CoverSnapshot {

label: result.label().to\_string(),

reason: result.label\_reason().map(|s| s.to\_string()),

snapshot\_vars: result.snapshot\_vars().iter().cloned().collect(),

registry\_name: result.registry\_name().to\_string(),

registry\_published\_date: result.registry\_published\_date().to\_string(),

}

}

fn map\_eligibility(result: &ResultDb) -> EligibilityBlock {

EligibilityBlock {

roll\_policy: result.roll\_policy\_pretty().to\_string(),

totals\_eligible\_roll: result.totals\_eligible\_roll(),

totals\_ballots\_cast: result.totals\_ballots\_cast(),

totals\_valid\_ballots: result.totals\_valid\_ballots(),

per\_unit\_quorum\_note: result.per\_unit\_quorum\_note(),

provenance: result.provenance().to\_string(),

}

}

fn map\_ballot(result: &ResultDb) -> BallotBlock {

let ballot\_type = result.ballot\_type().to\_string();

let approval\_sentence = ballot\_type.to\_ascii\_lowercase() == "approval";

BallotBlock {

ballot\_type,

allocation\_method: result.allocation\_method().to\_string(),

weighting\_method: result.weighting\_method().to\_string(),

approval\_denominator\_sentence: approval\_sentence,

}

}

fn map\_panel\_from\_gates(result: &ResultDb) -> LegitimacyPanel {

// Quorum

let (q\_num, q\_den) = result.gate\_quorum\_ratio();

let quorum = GateRow {

value\_pct\_1dp: pct\_1dp(q\_num, q\_den),

threshold\_pct\_0dp: pct0(result.gate\_quorum\_threshold\_pct()),

pass: result.gate\_quorum\_pass(),

denom\_note: None,

members\_hint: None,

};

// National approval majority (approval / valid ballots)

let (m\_num, m\_den) = result.gate\_majority\_ratio();

let majority = GateRow {

value\_pct\_1dp: pct\_1dp(m\_num, m\_den),

threshold\_pct\_0dp: pct0(result.gate\_majority\_threshold\_pct()),

pass: result.gate\_majority\_pass(),

denom\_note: Some("approval rate = approvals / valid ballots".to\_string()),

members\_hint: None,

};

// Double-majority (if enabled)

let dm = if result.gate\_double\_majority\_enabled() {

let (n\_num, n\_den) = result.gate\_double\_majority\_national\_ratio();

let (f\_num, f\_den) = result.gate\_double\_majority\_family\_ratio();

let th = pct0(result.gate\_double\_majority\_threshold\_pct());

let (pass\_nat, pass\_fam) = result.gate\_double\_majority\_pass\_both().unwrap\_or((false, false));

let hint = result.gate\_double\_majority\_family\_members();

let nat = GateRow {

value\_pct\_1dp: pct\_1dp(n\_num, n\_den),

threshold\_pct\_0dp: th.clone(),

pass: pass\_nat,

denom\_note: Some("approval rate = approvals / valid ballots".to\_string()),

members\_hint: None,

};

let fam = GateRow {

value\_pct\_1dp: pct\_1dp(f\_num, f\_den),

threshold\_pct\_0dp: th,

pass: pass\_fam,

denom\_note: Some("approval rate = approvals / valid ballots".to\_string()),

members\_hint: hint,

};

Some((nat, fam))

} else {

None

};

let symmetry = if result.symmetry\_enabled() { result.symmetry\_pass() } else { None };

// Overall pass/reasons mirror result label & gate passes (no recompute).

let pass = quorum.pass && majority.pass && dm.as\_ref().map\_or(true, |(n, f)| n.pass && f.pass)

&& symmetry.unwrap\_or(true);

let mut reasons: Vec<String> = Vec::new();

if !quorum.pass { reasons.push("Quorum failed".into()); }

if !majority.pass { reasons.push("National majority failed".into()); }

if let Some((n, f)) = &dm {

if !n.pass || !f.pass { reasons.push("Double-majority failed".into()); }

}

if let Some(false) = symmetry { reasons.push("Symmetry failed".into()); }

LegitimacyPanel { quorum, majority, double\_majority: dm, symmetry, pass, reasons }

}

fn map\_outcome\_from\_result(result: &ResultDb) -> OutcomeBlock {

OutcomeBlock {

label: result.label().to\_string(),

reason: result.label\_reason().unwrap\_or\_default().to\_string(),

national\_margin\_pp: pp\_signed(result.national\_margin\_pp()),

}

}

fn map\_frontier(fr: &FrontierMapDb, \_result: &ResultDb) -> FrontierBlock {

FrontierBlock {

mode: fr.mode\_pretty().to\_string(),

edge\_types: fr.edge\_types\_summary().to\_string(),

island\_rule: fr.island\_rule\_pretty().to\_string(),

bands\_summary: fr.bands\_summary().to\_vec(),

counters: fr.counters(),

}

}

fn map\_sensitivity(\_result: &ResultDb) -> Option<SensitivityBlock> {

// v1: no scenarios packaged ⇒ None

None

}

fn map\_integrity\_footer(

run: &RunRecordDb,

result: &ResultDb,

frontier: Option<&FrontierMapDb>,

) -> (IntegrityBlock, FooterIds) {

// Integrity

let (policy, seed\_opt) = (run.tie\_policy().to\_string(), run.tie\_seed\_opt());

let tie\_seed = match (policy.as\_str(), seed\_opt) {

("random", Some(s)) => Some(s.to\_string()),

\_ => None,

};

let integrity = IntegrityBlock {

engine\_vendor: run.engine\_vendor().to\_string(),

engine\_name: run.engine\_name().to\_string(),

engine\_version: run.engine\_version().to\_string(),

engine\_build: run.engine\_build().to\_string(),

formula\_id\_hex: run.formula\_id\_hex().to\_string(),

tie\_policy: policy,

tie\_seed,

started\_utc: run.started\_utc().to\_string(),

finished\_utc: run.finished\_utc().to\_string(),

};

// Footer IDs

let footer = FooterIds {

result\_id: result.result\_id().clone(),

run\_id: run.run\_id().clone(),

frontier\_id: frontier.map(|\_| result.frontier\_id()).flatten(),

reg\_id: result.reg\_id\_str().to\_string(),

param\_set\_id: result.param\_set\_id\_str().to\_string(),

tally\_id: result.tally\_id\_str().map(|s| s.to\_string()),

};

(integrity, footer)

}

// -------------------- Formatting helpers (integer math only) --------------------

fn pct\_1dp(num: i128, den: i128) -> String {

use vm\_core::rounding::percent\_one\_decimal\_tenths;

if den <= 0 { return "0.0%".into(); }

let tenths = percent\_one\_decimal\_tenths(num, den); // e.g., 553 → 55.3%

let sign = if tenths < 0 { "-" } else { "" };

let abs = tenths.abs();

let whole = abs / 10;

let dec = (abs % 10) as i128;

format!("{sign}{whole}.{dec}%")

}

fn pct0(value\_u8: u8) -> String {

format!("{value\_u8}%")

}

fn pp\_signed(pp\_i32: i32) -> String {

match pp\_i32.cmp(&0) {

core::cmp::Ordering::Greater => format!("+{pp\_i32} pp"),

core::cmp::Ordering::Equal => "±0 pp".to\_string(),

core::cmp::Ordering::Less => format!("{pp\_i32} pp"),

}

}

**Notes (alignment to your brief):**

* Sections and fields mirror Doc 7; lists preserve artifact order; keyed data uses deterministic containers only. One-decimal percentages are formatted **here** using integer math; renderers must not re-round. Approval ballots flip approval\_denominator\_sentence=true and add the fixed denominator note to majority rows. Gate rows are bound directly to precomputed ratios & thresholds (no recomputation). Integrity/footer copy engine info, FID, tie policy/seed (seed only when policy is random), and IDs echoed from artifacts.

This file expects the pipeline’s concrete artifact types to implement the three **view traits** (ResultView, RunRecordView, FrontierMapView). If you already have concrete structs, add thin wrappers or impl blocks to satisfy these traits and keep structure.rs decoupled from storage layout.

Sources

**You said:**

63 - crates vm\_report src render\_json.rs.md

File

code this

Thought for 1m 19s

**ChatGPT said:**

Implementation for crates/vm\_report/src/render\_json.rs, aligned to the serializer brief (deterministic Doc-7 order, no recomputation, omit optionals).

rust

CopyEdit

//! crates/vm\_report/src/render\_json.rs

//! Deterministic JSON renderer for ReportModel (Doc 7 fixed order). No I/O, no math, no hashing.

use serde\_json::{Map, Value};

use crate::structure::{

FrontierBlock, FrontierCounters, GateRow, ReportModel, SensitivityBlock,

};

/// Public API: return a compact UTF-8 JSON string (no trailing newline).

pub fn render\_json(model: &ReportModel) -> String {

let v = to\_ordered\_json(model);

// serde\_json::to\_string is deterministic for a given Value and Map insertion order.

serde\_json::to\_string(&v).expect("render\_json: serialization must not fail")

}

/// Internal: build a Value tree with \*\*stable insertion order\*\* for all objects.

fn to\_ordered\_json(m: &ReportModel) -> Value {

let mut root = Map::new();

// Insert sections strictly in Doc 7 order.

root.insert("cover".into(), cover\_json(m));

root.insert("eligibility".into(), eligibility\_json(m));

root.insert("ballot".into(), ballot\_json(m));

root.insert("legitimacy\_panel".into(), panel\_json(m));

root.insert("outcome".into(), outcome\_json(m));

if let Some(fr) = frontier\_json(m) {

root.insert("frontier".into(), fr);

}

if let Some(sens) = sensitivity\_json(m) {

root.insert("sensitivity".into(), sens);

}

root.insert("integrity".into(), integrity\_json(m));

root.insert("footer".into(), footer\_json(m));

Value::Object(root)

}

// ---------------- Section builders (fixed key ordering in each object) ----------------

fn cover\_json(m: &ReportModel) -> Value {

let mut obj = Map::new();

obj.insert("label".into(), Value::String(m.cover.label.clone()));

if let Some(r) = &m.cover.reason {

obj.insert("reason".into(), Value::String(r.clone()));

}

// snapshot\_vars: preserve model order

let vars = m

.cover

.snapshot\_vars

.iter()

.map(|kv| {

let mut kvobj = Map::new();

kvobj.insert("key".into(), Value::String(kv.key.clone()));

kvobj.insert("value".into(), Value::String(kv.value.clone()));

Value::Object(kvobj)

})

.collect::<Vec<\_>>();

obj.insert("snapshot\_vars".into(), Value::Array(vars));

obj.insert(

"registry\_name".into(),

Value::String(m.cover.registry\_name.clone()),

);

obj.insert(

"registry\_published\_date".into(),

Value::String(m.cover.registry\_published\_date.clone()),

);

Value::Object(obj)

}

fn eligibility\_json(m: &ReportModel) -> Value {

let mut obj = Map::new();

obj.insert(

"roll\_policy".into(),

Value::String(m.eligibility.roll\_policy.clone()),

);

obj.insert(

"totals\_eligible\_roll".into(),

Value::from(m.eligibility.totals\_eligible\_roll),

);

obj.insert(

"totals\_ballots\_cast".into(),

Value::from(m.eligibility.totals\_ballots\_cast),

);

obj.insert(

"totals\_valid\_ballots".into(),

Value::from(m.eligibility.totals\_valid\_ballots),

);

if let Some(note) = &m.eligibility.per\_unit\_quorum\_note {

obj.insert("per\_unit\_quorum\_note".into(), Value::String(note.clone()));

}

obj.insert(

"provenance".into(),

Value::String(m.eligibility.provenance.clone()),

);

Value::Object(obj)

}

fn ballot\_json(m: &ReportModel) -> Value {

let mut obj = Map::new();

obj.insert(

"ballot\_type".into(),

Value::String(m.ballot.ballot\_type.clone()),

);

obj.insert(

"allocation\_method".into(),

Value::String(m.ballot.allocation\_method.clone()),

);

obj.insert(

"weighting\_method".into(),

Value::String(m.ballot.weighting\_method.clone()),

);

// Emit the fixed approval denominator sentence flag only when true.

if m.ballot.approval\_denominator\_sentence {

obj.insert(

"approval\_denominator\_sentence".into(),

Value::Bool(true),

);

}

Value::Object(obj)

}

fn panel\_json(m: &ReportModel) -> Value {

let mut obj = Map::new();

obj.insert("quorum".into(), gate\_row(&m.panel.quorum));

obj.insert("majority".into(), gate\_row(&m.panel.majority));

if let Some((nat, fam)) = &m.panel.double\_majority {

let mut dm = Map::new();

dm.insert("national".into(), gate\_row(nat));

dm.insert("family".into(), gate\_row(fam));

obj.insert("double\_majority".into(), Value::Object(dm));

}

if let Some(sym) = m.panel.symmetry {

obj.insert("symmetry".into(), Value::Bool(sym));

}

obj.insert("pass".into(), Value::Bool(m.panel.pass));

// Reasons are already ordered by the builder; emit as-is.

let reasons = m

.panel

.reasons

.iter()

.cloned()

.map(Value::String)

.collect::<Vec<\_>>();

obj.insert("reasons".into(), Value::Array(reasons));

Value::Object(obj)

}

fn gate\_row(g: &GateRow) -> Value {

let mut obj = Map::new();

obj.insert(

"value\_pct\_1dp".into(),

Value::String(g.value\_pct\_1dp.clone()),

);

obj.insert(

"threshold\_pct\_0dp".into(),

Value::String(g.threshold\_pct\_0dp.clone()),

);

obj.insert("pass".into(), Value::Bool(g.pass));

if let Some(dn) = &g.denom\_note {

obj.insert("denom\_note".into(), Value::String(dn.clone()));

}

if let Some(hint) = &g.members\_hint {

let arr = hint.iter().cloned().map(Value::String).collect::<Vec<\_>>();

obj.insert("members\_hint".into(), Value::Array(arr));

}

Value::Object(obj)

}

fn outcome\_json(m: &ReportModel) -> Value {

let mut obj = Map::new();

obj.insert("label".into(), Value::String(m.outcome.label.clone()));

obj.insert("reason".into(), Value::String(m.outcome.reason.clone()));

obj.insert(

"national\_margin\_pp".into(),

Value::String(m.outcome.national\_margin\_pp.clone()),

);

Value::Object(obj)

}

fn frontier\_json(m: &ReportModel) -> Option<Value> {

let fr: &FrontierBlock = m.frontier.as\_ref()?;

let mut obj = Map::new();

obj.insert("mode".into(), Value::String(fr.mode.clone()));

obj.insert("edge\_types".into(), Value::String(fr.edge\_types.clone()));

obj.insert("island\_rule".into(), Value::String(fr.island\_rule.clone()));

// bands\_summary in declared order

let bands = fr

.bands\_summary

.iter()

.cloned()

.map(Value::String)

.collect::<Vec<\_>>();

obj.insert("bands\_summary".into(), Value::Array(bands));

// counters in a fixed key order

let c: &FrontierCounters = &fr.counters;

let mut counters = Map::new();

counters.insert("changed".into(), Value::from(c.changed));

counters.insert("no\_change".into(), Value::from(c.no\_change));

counters.insert("mediation".into(), Value::from(c.mediation));

counters.insert("enclave".into(), Value::from(c.enclave));

counters.insert(

"protected\_blocked".into(),

Value::from(c.protected\_blocked),

);

counters.insert("quorum\_blocked".into(), Value::from(c.quorum\_blocked));

obj.insert("counters".into(), Value::Object(counters));

Some(Value::Object(obj))

}

/// Policy: include `"sensitivity": "N/A (not executed)"` when absent.

fn sensitivity\_json(m: &ReportModel) -> Option<Value> {

match &m.sensitivity {

Some(SensitivityBlock { table\_2x3 }) => {

let rows = table\_2x3

.iter()

.map(|row| Value::Array(row.iter().cloned().map(Value::String).collect()))

.collect::<Vec<\_>>();

Some(Value::Object(

[("table\_2x3".into(), Value::Array(rows))].into\_iter().collect(),

))

}

None => Some(Value::String("N/A (not executed)".into())),

}

}

fn integrity\_json(m: &ReportModel) -> Value {

let mut obj = Map::new();

obj.insert(

"engine\_vendor".into(),

Value::String(m.integrity.engine\_vendor.clone()),

);

obj.insert(

"engine\_name".into(),

Value::String(m.integrity.engine\_name.clone()),

);

obj.insert(

"engine\_version".into(),

Value::String(m.integrity.engine\_version.clone()),

);

obj.insert(

"engine\_build".into(),

Value::String(m.integrity.engine\_build.clone()),

);

obj.insert(

"formula\_id\_hex".into(),

Value::String(m.integrity.formula\_id\_hex.clone()),

);

obj.insert(

"tie\_policy".into(),

Value::String(m.integrity.tie\_policy.clone()),

);

if let Some(seed) = &m.integrity.tie\_seed {

obj.insert("tie\_seed".into(), Value::String(seed.clone()));

}

obj.insert(

"started\_utc".into(),

Value::String(m.integrity.started\_utc.clone()),

);

obj.insert(

"finished\_utc".into(),

Value::String(m.integrity.finished\_utc.clone()),

);

Value::Object(obj)

}

fn footer\_json(m: &ReportModel) -> Value {

let mut obj = Map::new();

// Strong types display as canonical strings via Display

obj.insert("result\_id".into(), Value::String(m.footer.result\_id.to\_string()));

obj.insert("run\_id".into(), Value::String(m.footer.run\_id.to\_string()));

if let Some(fr) = &m.footer.frontier\_id {

obj.insert("frontier\_id".into(), Value::String(fr.to\_string()));

}

obj.insert("reg\_id".into(), Value::String(m.footer.reg\_id.clone()));

obj.insert(

"param\_set\_id".into(),

Value::String(m.footer.param\_set\_id.clone()),

);

if let Some(tly) = &m.footer.tally\_id {

obj.insert("tally\_id".into(), Value::String(tly.clone()));

}

Value::Object(obj)

}

#[cfg(test)]

mod tests {

use super::\*;

use crate::structure::\*;

fn demo\_model() -> ReportModel {

ReportModel {

cover: CoverSnapshot {

label: "Decisive".into(),

reason: Some("All gates passed".into()),

snapshot\_vars: vec![SnapshotVar{ key:"VAR-001".into(), value:"plurality".into() }],

registry\_name: "Demo Registry".into(),

registry\_published\_date: "2025-01-01".into(),

},

eligibility: EligibilityBlock {

roll\_policy: "Resident roll".into(),

totals\_eligible\_roll: 1000,

totals\_ballots\_cast: 700,

totals\_valid\_ballots: 680,

per\_unit\_quorum\_note: None,

provenance: "demo v1".into(),

},

ballot: BallotBlock {

ballot\_type: "approval".into(),

allocation\_method: "wta".into(),

weighting\_method: "none".into(),

approval\_denominator\_sentence: true,

},

panel: LegitimacyPanel {

quorum: GateRow {

value\_pct\_1dp: "70.0%".into(),

threshold\_pct\_0dp: "50%".into(),

pass: true,

denom\_note: None,

members\_hint: None,

},

majority: GateRow {

value\_pct\_1dp: "55.0%".into(),

threshold\_pct\_0dp: "55%".into(),

pass: true,

denom\_note: Some("approval rate = approvals / valid ballots".into()),

members\_hint: None,

},

double\_majority: None,

symmetry: None,

pass: true,

reasons: vec![],

},

outcome: OutcomeBlock {

label: "Decisive".into(),

reason: "All gates passed".into(),

national\_margin\_pp: "+10 pp".into(),

},

frontier: None,

sensitivity: None,

integrity: IntegrityBlock {

engine\_vendor: "KOA".into(),

engine\_name: "VM".into(),

engine\_version: "0.1.0".into(),

engine\_build: "abc123".into(),

formula\_id\_hex: "0".repeat(64),

tie\_policy: "deterministic".into(),

tie\_seed: None,

started\_utc: "2025-08-12T14:00:00Z".into(),

finished\_utc: "2025-08-12T14:00:05Z".into(),

},

footer: FooterIds {

result\_id: "RES:0123456789abcdef0123456789abcdef0123456789abcdef0123456789abcd".parse().unwrap(),

run\_id: "RUN:2025-08-12T14:00:00Z-0123456789abcdef0123456789abcdef0123456789abcdef0123456789abcd".parse().unwrap(),

frontier\_id: None,

reg\_id: "REG:demo".into(),

param\_set\_id: "PS:demo".into(),

tally\_id: None,

},

}

}

#[test]

fn deterministic\_and\_in\_order() {

let m = demo\_model();

let s1 = render\_json(&m);

let s2 = render\_json(&m);

assert\_eq!(s1, s2);

// Top-level order sanity (cover → eligibility → ballot → legitimacy\_panel → outcome → integrity → footer)

let cover\_pos = s1.find("\"cover\"").unwrap();

let elig\_pos = s1.find("\"eligibility\"").unwrap();

let ballot\_pos = s1.find("\"ballot\"").unwrap();

let panel\_pos = s1.find("\"legitimacy\_panel\"").unwrap();

let outcome\_pos = s1.find("\"outcome\"").unwrap();

let integ\_pos = s1.find("\"integrity\"").unwrap();

let footer\_pos = s1.find("\"footer\"").unwrap();

assert!(cover\_pos < elig\_pos && elig\_pos < ballot\_pos && ballot\_pos < panel\_pos);

assert!(panel\_pos < outcome\_pos && outcome\_pos < integ\_pos && integ\_pos < footer\_pos);

// Approval sentence present

assert!(s1.contains("\"approval\_denominator\_sentence\":true"));

}

}

**Why this matches the brief (concise):**

* Keys are inserted in Doc-7 order and each section’s fields are added in a fixed sequence; optionals (reason, double\_majority, symmetry, frontier, tie\_seed, tally\_id) are omitted when absent.
* All numerics (percents, pp) are passed through verbatim from the ReportModel; there’s zero recomputation.
* Output is a compact, deterministic JSON string (UTF-8, no trailing newline).

Sources

**You said:**

66 - crates vm\_cli src args.rs.md

File

code this

Thought for 1m 17s

**ChatGPT said:**

Implementation for crates/vm\_cli/src/args.rs, matching the CLI contract (mode gating, local paths only, quick manifest check, seed parsing).

rust

CopyEdit

//! crates/vm\_cli/src/args.rs

//! Deterministic, offline CLI surface for the VM pipeline.

//! - Two modes: manifest OR explicit paths

//! - Exactly one of --ballots | --tally in explicit mode

//! - Local files only (no http/https)

//! - Optional --seed as decimal u64 or 0x-hex (≤16 hex digits)

//!

//! This module only validates and normalizes arguments. It does not run I/O-heavy schema checks.

use std::env;

use std::fs;

use std::path::{Path, PathBuf};

use clap::Parser;

use serde\_json::{self as sj, Value};

use thiserror::Error;

/// CLI argument model (parse-only; validation happens in `parse\_and\_validate`).

#[derive(Debug, Parser)]

#[command(name = "vm", disable\_help\_subcommand = true)]

pub struct Args {

// ---- Mode selection ----

#[arg(long, conflicts\_with\_all = ["registry", "params", "ballots", "tally", "adjacency", "autonomy"])]

pub manifest: Option<PathBuf>,

// ---- Explicit mode ----

#[arg(long)]

pub registry: Option<PathBuf>,

#[arg(long)]

pub params: Option<PathBuf>,

#[arg(long, conflicts\_with = "tally")]

pub ballots: Option<PathBuf>,

#[arg(long, conflicts\_with = "ballots")]

pub tally: Option<PathBuf>,

// ---- Optional inputs ----

#[arg(long)]

pub adjacency: Option<PathBuf>,

#[arg(long)]

pub autonomy: Option<PathBuf>,

// ---- Output & rendering ----

#[arg(long, default\_value = ".")]

pub out: PathBuf,

#[arg(long, value\_parser = ["json", "html"], num\_args = 0..=2)]

pub render: Vec<String>,

// ---- Determinism & control ----

/// Optional override for tie RNG seed; decimal u64 or 0x-hex (≤16 hex digits).

#[arg(long)]

pub seed: Option<String>,

#[arg(long)]

pub validate\_only: bool,

#[arg(long)]

pub quiet: bool,

}

#[derive(Error, Debug)]

pub enum CliError {

#[error("invalid flag combination: {0}")]

BadCombo(&'static str),

#[error("missing required flag: {0}")]

Missing(&'static str),

#[error("both or neither of --ballots/--tally provided")]

BallotsTallyChoice,

#[error("path must be local file (no scheme): {0}")]

NonLocalPath(String),

#[error("file not found: {0}")]

NotFound(String),

#[error("invalid seed: {0}")]

BadSeed(String),

#[error("manifest quick-check failed: {0}")]

ManifestQuick(String),

}

/// Parse argv, validate combinations, ensure local files exist, normalize paths,

/// quick-check manifests, and validate seed. Returns sanitized Args.

pub fn parse\_and\_validate() -> Result<Args, CliError> {

let mut a = Args::parse();

// Rendering default: if user didn't pass any, default to ["json"].

if a.render.is\_empty() {

a.render.push("json".to\_string());

}

// Normalize `out` immediately (it may not exist yet; don't error)

a.out = normalize\_path(&a.out);

// Mode validation

if a.manifest.is\_some() {

validate\_manifest\_mode(&a)?;

// Ensure manifest file exists and quick-check contents

let man\_path = normalize\_path(a.manifest.as\_ref().unwrap());

ensure\_local\_exists(&man\_path, "manifest")?;

// Perform quick JSON shape checks (no schema, no hashing)

let bytes = fs::read(&man\_path).map\_err(|\_| CliError::NotFound(man\_path.display().to\_string()))?;

quick\_check\_manifest\_bytes(&bytes)?;

// Store normalized manifest path back

a.manifest = Some(man\_path);

} else {

validate\_explicit\_mode(&a)?;

// Normalize and check explicit paths

a.registry = Some(normalize\_and\_check(a.registry.take().unwrap(), "registry")?);

a.params = Some(normalize\_and\_check(a.params.take().unwrap(), "params")?);

match (a.ballots.take(), a.tally.take()) {

(Some(p), None) => a.ballots = Some(normalize\_and\_check(p, "ballots")?),

(None, Some(p)) => a.tally = Some(normalize\_and\_check(p, "tally")?),

\_ => return Err(CliError::BallotsTallyChoice),

}

if let Some(adj) = a.adjacency.take() {

a.adjacency = Some(normalize\_and\_check(adj, "adjacency")?);

}

if let Some(pkg) = a.autonomy.take() {

a.autonomy = Some(normalize\_and\_check(pkg, "autonomy")?);

}

}

// Seed validation (convertible to u64)

if let Some(s) = &a.seed {

let \_ = parse\_seed\_u64(s)?; // validate now; main() can parse again to get u64

}

Ok(a)

}

// ------------------------ Validation helpers ------------------------

pub fn validate\_manifest\_mode(a: &Args) -> Result<(), CliError> {

// Conflicts are already enforced by clap attributes; just ensure presence.

if a.manifest.is\_none() {

return Err(CliError::Missing("--manifest"));

}

Ok(())

}

pub fn validate\_explicit\_mode(a: &Args) -> Result<(), CliError> {

if a.registry.is\_none() {

return Err(CliError::Missing("--registry"));

}

if a.params.is\_none() {

return Err(CliError::Missing("--params"));

}

// Exactly one of ballots | tally

match (&a.ballots, &a.tally) {

(Some(\_), None) | (None, Some(\_)) => {}

\_ => return Err(CliError::BallotsTallyChoice),

}

Ok(())

}

fn normalize\_and\_check(p: PathBuf, label: &'static str) -> Result<PathBuf, CliError> {

let p = normalize\_path(&p);

ensure\_local\_exists(&p, label)?;

Ok(p)

}

pub fn ensure\_local\_exists(p: &PathBuf, label: &'static str) -> Result<(), CliError> {

if looks\_like\_url(&p) {

return Err(CliError::NonLocalPath(format!("{label}={}", p.display())));

}

match fs::metadata(p) {

Ok(md) if md.is\_file() => Ok(()),

\_ => Err(CliError::NotFound(format!("{label}={}", p.display()))),

}

}

pub fn normalize\_path(p: &PathBuf) -> PathBuf {

// Best-effort canonicalize; if that fails (non-existent), absolutize with CWD.

match fs::canonicalize(p) {

Ok(abs) => abs,

Err(\_) => {

// If path is already absolute, just clone; else join with current dir.

if p.is\_absolute() {

p.clone()

} else {

env::current\_dir().unwrap\_or\_else(|\_| PathBuf::from(".")).join(p)

}

}

}

}

// ------------------------ Seed parsing ------------------------

/// Accept decimal `\d+` or 0xHEX (1..=16 hex digits) → u64.

pub fn parse\_seed\_u64(s: &str) -> Result<u64, CliError> {

let t = s.trim();

if let Some(hex) = t.strip\_prefix("0x").or\_else(|| t.strip\_prefix("0X")) {

if hex.is\_empty() || hex.len() > 16 || !hex.chars().all(|c| c.is\_ascii\_hexdigit()) {

return Err(CliError::BadSeed(t.to\_string()));

}

u64::from\_str\_radix(hex, 16).map\_err(|\_| CliError::BadSeed(t.to\_string()))

} else {

// Decimal

t.parse::<u64>().map\_err(|\_| CliError::BadSeed(t.to\_string()))

}

}

// ------------------------ Manifest quick-check ------------------------

/// Lightweight manifest shape check:

/// - require reg\_path & params\_path (strings)

/// - require exactly one of ballots\_path XOR ballot\_tally\_path (string)

/// - reject http(s):// in any path-like field

/// - if "digests" present, any sha256 must be 64 hex chars (case-insensitive)

pub fn quick\_check\_manifest\_bytes(bytes: &[u8]) -> Result<(), CliError> {

let v: Value = sj::from\_slice(bytes)

.map\_err(|e| CliError::ManifestQuick(format!("invalid JSON: {e}")))?;

let obj = v.as\_object().ok\_or\_else(|| CliError::ManifestQuick("root is not a JSON object".into()))?;

// Required fields

let reg = obj.get("reg\_path").and\_then(Value::as\_str).ok\_or\_else(|| CliError::ManifestQuick("missing reg\_path".into()))?;

let par = obj.get("params\_path").and\_then(Value::as\_str).ok\_or\_else(|| CliError::ManifestQuick("missing params\_path".into()))?;

// ballots\_path XOR ballot\_tally\_path

let b\_legacy = obj.get("ballots\_path").and\_then(Value::as\_str);

let b\_tally = obj.get("ballot\_tally\_path").and\_then(Value::as\_str);

let one\_of = matches!((b\_legacy, b\_tally), (Some(\_), None) | (None, Some(\_)));

if !one\_of {

return Err(CliError::ManifestQuick(

"exactly one of ballots\_path | ballot\_tally\_path is required".into(),

));

}

// Local-only paths

for (label, s\_opt) in [

("reg\_path", Some(reg)),

("params\_path", Some(par)),

("ballots\_path", b\_legacy),

("ballot\_tally\_path", b\_tally),

("adjacency\_path", obj.get("adjacency\_path").and\_then(Value::as\_str)),

("autonomy\_path", obj.get("autonomy\_path").and\_then(Value::as\_str)),

] {

if let Some(s) = s\_opt {

if looks\_like\_url\_str(s) {

return Err(CliError::ManifestQuick(format!("{label} must be local (no URLs)")));

}

}

}

// Optional digests shape

if let Some(digests) = obj.get("digests").and\_then(Value::as\_object) {

for (k, v) in digests {

if let Some(sha) = v.get("sha256").and\_then(Value::as\_str) {

if !is\_hex64(sha) {

return Err(CliError::ManifestQuick(format!("digests[{k}].sha256 must be 64 hex chars")));

}

} else {

return Err(CliError::ManifestQuick(format!("digests[{k}].sha256 missing")));

}

}

}

Ok(())

}

// ------------------------ Small utilities ------------------------

fn looks\_like\_url(p: &Path) -> bool {

if let Some(s) = p.to\_str() {

looks\_like\_url\_str(s)

} else {

false

}

}

fn looks\_like\_url\_str(s: &str) -> bool {

let sl = s.trim().to\_ascii\_lowercase();

sl.starts\_with("http://") || sl.starts\_with("https://")

}

fn is\_hex64(s: &str) -> bool {

s.len() == 64 && s.bytes().all(|b| (b'0'..=b'9').contains(&b) || (b'a'..=b'f').contains(&b) || (b'A'..=b'F').contains(&b))

}

// ------------------------ Tests ------------------------

#[cfg(test)]

mod tests {

use super::\*;

#[test]

fn seed\_decimal\_and\_hex() {

assert\_eq!(parse\_seed\_u64("123").unwrap(), 123);

assert\_eq!(parse\_seed\_u64("0xFF").unwrap(), 255);

assert!(parse\_seed\_u64("0x").is\_err());

assert!(parse\_seed\_u64("0x1234567890ABCDEF1").is\_err()); // >16 hex digits

assert!(parse\_seed\_u64("-1").is\_err());

}

#[test]

fn manifest\_quick\_check\_variants() {

let ok = br#"{

"reg\_path":"reg.json",

"params\_path":"params.json",

"ballot\_tally\_path":"tally.json",

"digests": { "params\_path": { "sha256": "0123456789abcdef0123456789abcdef0123456789abcdef0123456789abcd" } }

}"#;

assert!(quick\_check\_manifest\_bytes(ok).is\_ok());

let both = br#"{"reg\_path":"r","params\_path":"p","ballots\_path":"x","ballot\_tally\_path":"y"}"#;

assert!(quick\_check\_manifest\_bytes(both).is\_err());

let none = br#"{"reg\_path":"r","params\_path":"p"}"#;

assert!(quick\_check\_manifest\_bytes(none).is\_err());

let url = br#"{"reg\_path":"http://x","params\_path":"p","ballot\_tally\_path":"t"}"#;

assert!(quick\_check\_manifest\_bytes(url).is\_err());

let badsha = br#"{"reg\_path":"r","params\_path":"p","ballot\_tally\_path":"t","digests":{"reg\_path":{"sha256":"zzz"}}}"#;

assert!(quick\_check\_manifest\_bytes(badsha).is\_err());

}

#[test]

fn normalize\_and\_local\_checks() {

let cwd = env::current\_dir().unwrap();

let rel = PathBuf::from("Cargo.toml");

let abs = normalize\_path(&rel);

assert!(abs.is\_absolute());

// Local-only check (won't exist in tests, so only test url rejection branch)

assert!(ensure\_local\_exists(&PathBuf::from("http://x"), "x").is\_err());

}

#[test]

fn explicit\_mode\_requirements() {

let mut a = Args::parse\_from(["vm","--registry","r","--params","p","--tally","t"]);

a.render.clear();

assert!(validate\_explicit\_mode(&a).is\_ok());

// ballots & tally both -> error

let a = Args::parse\_from(["vm","--registry","r","--params","p","--ballots","b","--tally","t"]);

assert!(validate\_explicit\_mode(&a).is\_err());

}

}

Sources

**You said:**

this is the content of which file: [package] name = "vm\_cli" version = "0.1.0" edition = "2021" license = "Apache-2.0 OR MIT" resolver = "2" description = "Offline, deterministic CLI for the VM engine (Docs 4–5 pipeline)" # no build.rs [[bin]] name = "vm" path = "src/main.rs" [features] # Defaults: keep things deterministic and offline; JSON reporting available by default. default = ["std", "report-json"] # Minimal std passthrough (propagate to vm\_report so it can enable std when linked from the CLI) std = ["vm\_report/std"] # Reporting frontends (map 1:1 to vm\_report features; pull serde only for JSON path) report-json = ["vm\_report/render\_json", "dep:serde", "dep:serde\_json"] report-html = ["vm\_report/render\_html"] # Frontier support toggles (vm\_pipeline controls the step; vm\_report has no extra frontier feature) frontier = ["vm\_pipeline/frontier"] [dependencies] vm\_pipeline = { path = "../vm\_pipeline" } vm\_io = { path = "../vm\_io" } vm\_report = { path = "../vm\_report", default-features = false } clap = { version = "4", features = ["derive"], default-features = false } # Optional pretty/JSON printing for CLI flags serde = { version = "1", features = ["derive"], optional = true, default-features = false } serde\_json = { version = "1", optional = true } # Optional progress UI; keep off by default to avoid nondeterministic terminal probing indicatif = { version = "0.17", optional = true, default-features = false } [dev-dependencies] assert\_cmd = "2" predicates = "3" [profile.release] lto = true codegen-units = 1 panic = "abort" strip = "symbols"

**ChatGPT said:**

Thought for 16s

crates/vm\_cli/Cargo.toml

**You said:**

"C:\MonCode\VotingMachine\voting-machine\.github\workflows\ci.yml" "C:\MonCode\VotingMachine\voting-machine\ci\perf\_profile.json" "C:\MonCode\VotingMachine\voting-machine\ci\determinism.yml" What are those files. Have we forgot to code them. do not code, just answer my question

**ChatGPT said:**

These three files are **not Rust source files from your crates** — they’re project-level configuration and CI tooling files:

1. **.github/workflows/ci.yml**
   * GitHub Actions workflow definition.
   * Describes your continuous integration pipeline (build, test, lint, possibly determinism checks) triggered on pushes/PRs.
   * Not part of the Rust crate codebase itself, but it’s critical for automated testing and reproducibility.
2. **ci/perf\_profile.json**
   * JSON profile for performance testing.
   * Likely defines benchmark scenarios, input datasets, or parameter sets for profiling your VM engine.
   * Used by CI or manual scripts to measure runtime and efficiency.
3. **ci/determinism.yml**
   * CI configuration for determinism verification.
   * Probably runs a special pipeline to ensure identical outputs are produced given the same inputs across OS/architectures, as required by your specs.

They don’t need to be “coded” like the 89 crate files we’re working through — they are already config assets that should just be present in the repo. If they’re missing, you’d need to recreate them, but they are not part of the Rust coding alignment process.