The Vela Protocol Specification

Will Blair
constellate.science/specification
Technical companion to The Constellate Architecture · v0.5 · May 26, 2026

Abstract

This is the technical specification for the Vela protocol — the substrate layer of the Constellate architecture. The Constellate Architecture whitepaper (v0.1) makes the case for *why* the substrate matters; this document specifies *what* the substrate is, at the level of detail an implementer needs.

The specification is normative. Where the whitepaper describes mechanics at the architectural level, this document defines the bit-level shapes: type prefixes, canonical event structure, reducer mutation kinds, canonicalization rules, signature semantics, and cross-implementation conformance criteria. Two reducer implementations that follow this specification will produce byte-identical state from the same event log.

The version tracks the Vela protocol release rather than the whitepaper's architectural version. Spec v0.5 corresponds to Vela protocol v0.500 and Carina kernel v0.6. Future revisions will accompany kernel-version increments under the schema-evolution process described in §8.

1 Introduction

Vela is the protocol layer of Constellate. Its core commitments are three: replay determinism (a per-frontier event log that two independent reducer implementations materialize identically), signed state transitions as the canonical object (not artifacts, not metadata, not relationships), and structural forkability (history travels with the fork so capture at any orchestration layer fails to entrap the underlying state).

The protocol is implemented in Rust (reference, at `crates/vela-protocol/`), Python (at ~78% conformance against the test vector suite, with gaps documented in §9), and TypeScript (at ~41% conformance, gaps documented in §9). Vela ships at v0.500+ under dual Apache-2.0 / MIT licensing at `github.com/vela-science/vela`. The public release at v0.48.0 (2026-05-02) is the canonical reference; v0.500+ is the active development line.

This specification covers the bit-level mechanics. Operational details — running a peer hub, deploying a reducer, hosting a corridor — live in the repository's `docs/` directory. Pilot-deployment details live in the companion *First Corridor Pilot Plan v0.1*.

2 Type System Primitives

The Carina kernel defines stable type prefixes for all first-class objects. The current set at Carina v0.6:

Prefix	Object	Carina Schema
vf_	Finding	vela.finding.v0
vfr_	Frontier	(meta; derived from genesis event)
vbr_	Bridge	candidate cross-frontier link
vat_	Atlas	vela.atlas.v0.1
vco_	Constellation	vela.constellation.v0.1 (reserved)
vpr_	Proposal	ephemeral working object
vev_	Event	vela.event.v0.1
ve_ / vea_	Evidence / Evidence Atom	v0
vcnd_	Condition Record	v0
vnr_	Negative Result	vela.negative-result.v0
vtr_	Trajectory	vela.trajectory.v0
va_	Artifact	vela.artifact.v0
vrep_	Replication	vela.replication.v0
vpred_	Prediction	vela.prediction.v0
vres_	Resolution	vela.resolution.v0
vd_	Dataset	vela.dataset.v0
vc_	Code Artifact	vela.code-artifact.v0
vsd_	Diff Pack	vela.diff-pack.v0
vs_	Source	(materialized in projections)
vcj_	Conjecture	vela.conjecture.v0
pp_	Proof Packet	vela.proof-packet.v0
vgp_	Registry Governance Policy	vela.registry-governance-policy.v0.1

Identifiers are content-addressed where derived from object content. Two independent observers of the same canonical object produce the same identifier. Implementations must reject events referencing types not in the current kernel version or events where the prefix does not match the declared schema.

3 The Finding Bundle

The finding bundle is the primary object Vela holds. Each bundle carries a bounded scientific claim with the evidence and provenance that support it.

A bundle has a stable identifier of the form vf_ followed by the full hex encoding of a SHA-256 digest computed over the normalized assertion text, assertion type, and provenance identifier — 64 hex characters, 256 bits of collision resistance. The provenance identifier prefers DOI, then PubMed ID, then source title. Surface contexts (UI, references in prose, conflict-detected event payloads) typically display the leading 16 hex characters as an abbreviation, but the canonical identifier is the full digest. Two systems that observe the same claim from the same source produce the same canonical identifier; abbreviations are display-only and resolve to the canonical form. The full-width choice follows Git's evolution from 7-character to 12-character to full-40-character commit identifiers as the codebase scaled, and ensures that a substrate

intended to outlast decades plus agent-rate proposals retains a collision budget well above the birthday-attack threshold.

A finding bundle holds:

- The assertion text and assertion type
- The evidence atoms (`vea_*`) anchored to source artifacts
- The condition records (`vcnd_*`) defining scope
- The trajectory (`vtr_*`) of intermediate steps
- The confidence model (Beta distribution at v0.5; credal sets under design for v0.7)
- The dependencies on prior findings
- The signatures from credentialed reviewers
- The access tier (Public, Restricted, Classified)
- Lifecycle flags (asserted, reviewed, caveated, retracted)

The bundle is immutable in its identity. State changes enter through canonical events that mutate the bundle's collection fields (signatures, evidence atoms, condition records) and lifecycle flags; the identity hash never changes.

4 Frontiers

A **frontier** (`vfr_*`) is a bounded, reviewable scientific question — the unit of replay determinism and federation. A frontier's identifier is derived from the SHA-256 of its genesis event preimage; the genesis event is the structural anchor that every subsequent event chains from.

Frontiers carry the per-frontier event log, the pinned Carina kernel digest, the manifest of participating actors and reviewers, and the registry governance policy (`vgp_*`). Two reducer implementations replaying the same event log under the same pinned kernel digest produce byte-identical finding-state digests.

Federation moves state at frontier granularity. A peer hub mirrors a frontier — not an Atlas, not a Constellation — because frontier-level integrity is what the protocol can guarantee. Cross-frontier dependencies are tracked as explicit references (§11) with snapshot hashes pinning the target's state at reference time.

5 Canonical Event Shape

Fields marked [**shipped**] are present in the Vela v0.500 `StateEvent` struct at `crates/vela-protocol/src/events.rs`. Fields marked [**spec-only**] are pinned here as targets for Carina v0.7 schema evolution and are not yet present in the shipped envelope. The `signature: Option<String>` field shipped today is the v0.500 single-signature form; the multi-signature `signatures: Vec<Signature>` shape in this specification is the target shape under §7.3 quorum semantics.

```
pub struct StateEvent {
    pub id: String,                      // vev_*                          [shipped]
    pub schema_version: u32,             // Carina event-schema version    [spec-
only — currently `schema: String`]
    pub kernel_digest: String,           // sha256 of Carina kernel        [spec-
only]
    pub frontier: String,                // vfr_* — signature replay-binding [spec-
only]
```

```
    pub kind: String,                    // e.g., "finding.reviewed"       [shipped]
    pub target: EventTarget,             // {type, id}                     [shipped]
    pub parent_event_id: Option<String>, // vev_* — per-target chain link  [spec-
only]
    pub actor: ActorRef,                 // {id, type, tier}               [shipped
— `tier` field spec-only]
    pub actor_sequence: u64,             // monotonic per actor            [spec-
only]
    pub timestamp: String,               // RFC3339                        [shipped]
    pub before_hash: String,             // sha256 of state before         [shipped]
    pub after_hash: String,              // sha256 of state after          [shipped]
    pub reason: String,                  // reviewer justification         [shipped]
    pub payload: serde_json::Value,      // kind-specific data             [shipped]
    pub signatures: Vec<Signature>,      // Ed25519 sigs over canonical    [spec-
only — currently `signature: Option<String>`]
    pub witness_set_version: Option<u32>, // governance events; per §8.2    [spec-
only]
}

pub struct Signature {
    pub actor_id: String,
    pub public_key: String,              // hex-encoded Ed25519 key used to sign
    pub algorithm: String,               // "ed25519"
    pub signed_at: String,               // RFC3339
    pub signature: String,               // hex-encoded 64-byte Ed25519 signature
}
```

The envelope carries five fields whose absence would otherwise force protocol guarantees to live implicitly in surrounding prose:

- **`schema_version`** (u32) — names the Carina event-schema version the event was authored under, so a reducer encountering an event from a newer kernel can either reject it or apply documented migration semantics rather than silently misinterpret unknown fields. Schema evolution operates at this version granularity.
- **`kernel_digest`** (string) — the SHA-256 digest of the exact Carina kernel definition the event was authored against. A reducer materializes state against this pinned kernel; the digest must match a transparency-log-published kernel release (§8.1) or the event is rejected under strict mode.
- **`frontier`** (string) — the `vfr_*` identifier of the containing frontier. Because the field is part of the canonicalized payload the signatures cover, an Ed25519 signature on this event cannot be replayed against any other frontier. Cross-frontier signature replay is closed structurally rather than procedurally.
- **`parent_event_id`** (optional `vev_*`) — explicit per-target chain link to the prior event on the same target finding. The chain was previously implicit (recoverable from before_hash/ after_hash matching); making it explicit lets a reducer verify the chain without scanning the global event log for the prior after_hash.
- **`actor_sequence`** (u64) — monotonic per-actor counter, preventing an adversary who captures a signed event from replaying it as a new submission. Each actor's sequence number must strictly increase across their signed events within a frontier. The sequence is anchored to the actor's stable identity, not to any individual signing key, so it continues across key rotations — the rotation event itself carries the prior sequence number, and the post-rotation

event uses the next value. This means an actor's full lifetime of signed events forms a single monotonic chain per frontier regardless of how many keys they have used.

- **`witness_set_version`** (u32, governance events only) — for governance events, identifies which witness set (per §8.2) is required to cosign. Historical events under prior witness sets remain valid forever; the version pin lets the reducer accept the historical cosignatures without requiring the historical witness set to still be operating.

Replay requires `before_hash` of event N on a target finding to equal `after_hash` of event N-1 on the same finding, the `parent_event_id` of event N to match the `id` of event N-1 on the same target, and the `actor_sequence` of each actor to strictly increase across their events. The chain is per-finding; chains for different findings interleave in the global event log but are validated independently per target. The canonicalized payload over which `signatures` is computed is defined in §10.

6 Reducer Mutation Kinds

The reducer is a pure function from `(state, event) → state`. It dispatches on the event's kind and mutates the frontier state deterministically.

The `REDUCER_MUTATION_KINDS` constant at `crates/vela-protocol/src/reducer.rs` enumerates the 26 kinds whose dispatch arms mutate finding, negative-result, trajectory, artifact, or evidence-atom state. The list at Vela v0.500 (Carina v0.6):

```
finding.asserted, finding.reviewed, finding.noted, finding.caveated,
finding.confidence_revised, finding.rejected, finding.retracted,
finding.dependency_invalidated, finding.span_repaired,
finding.entity_resolved, finding.entity_added,
negative_result.asserted, negative_result.reviewed, negative_result.retracted,
trajectory.created, trajectory.step_appended, trajectory.reviewed,
trajectory.retracted,
artifact.asserted, artifact.reviewed, artifact.retracted,
tier.set,
evidence_atom.locator_repaired,
diff_pack.released, diff_pack.reviewed,
verdict_conflict.resolved
```

A small number of additional kinds are dispatched by `apply_event` but excluded from `REDUCER_MUTATION_KINDS` because they do not mutate any of the indexed object collections: the federation-observation kinds (`frontier.synced_with_peer`, `frontier.conflict_detected`, `frontier.conflict_resolved`), the genesis kind `frontier.created`, `attestation.recorded`, `replication.deposited`, and `prediction.deposited`. These appear in event logs and are accepted by the reducer; their state effects (where any) live on collections orthogonal to the canonical finding-state digest.

The reducer treats unknown kinds as errors under strict mode and as no-ops under permissive mode. New kinds are added through schema evolution under Carina version increments rather than as silent additions.

7 Identity, Signing, and the Agent-Attestation Tier

7.1 Actor Records

Vela's identity model is built around Ed25519 keypairs tied to registered actor records, with optional ORCID anchoring, access tiers, and revocation.

An actor record carries a stable namespaced identifier (`actor/<institution>/<handle>`), the actor's Ed25519 public key(s), an optional ORCID, the institutional affiliation, the actor's role (researcher, reviewer, lab, agent, institution, system), the access-tier classification (Public, Restricted, Classified), and lifecycle metadata (registered_at, revoked_at).

Actor identity is anchored to the stable identifier, not to any particular signing key. Key rotation (§7.5) replaces the active signing key without changing the actor's identity; revocation of a compromised key invalidates new signatures from that key but does not retroactively void prior signatures the key produced.

7.2 Signatures

All signatures use Ed25519 over the canonicalized payload defined in §10. Each event's `signatures` field carries one or more `Signature` records, each binding an actor identifier, a public key, a signing timestamp, and the signature bytes.

The canonicalized payload for signature production excludes the `signatures` field itself (the self-reference rule, §10) and includes the `frontier` field (the cross-frontier replay-binding rule, §5). A signature on an event is valid only against the specific frontier the event names.

7.3 Signature Thresholds

A finding may carry a signature-threshold flag requiring k distinct valid signatures from registered actors before the finding is accepted into canonical state. The threshold is set per-finding by the depositor or by corridor policy. Findings with safety-relevant, regulator-bound, or high-dependency scope typically carry thresholds of 2-3; low-stakes deposits typically carry threshold 1.

Signature thresholds compose with the corridor's Registry Governance Policy (`vgp_*`): the policy specifies which actor roles can sign for which event kinds, and the threshold enforces how many distinct signers are required. Threshold-not-met events sit in the reviewer queue; they do not enter canonical state.

7.4 Revocation

Revocation is recorded as an event amending an actor's record with a `revoked_at` timestamp and a reason. Signatures produced before the revocation timestamp remain valid for the historical record; new signatures from the revoked key are rejected. Past acceptances under a compromised key are not retroactively voided — they remain inspectable, and downstream consumers can decide whether to re-review claims that depend on the compromised actor.

7.5 Key Rotation

Key rotation is a protocol-recognized event: an actor's record is amended to add a new public key with a `valid_from` timestamp, and the prior key receives a `valid_until` timestamp (which

may be in the future to allow overlap). The rotation event itself is signed under the prior key; the post-rotation event uses the new key. Both signatures verify against the actor's record at the rotation timestamp.

The actor's `actor_sequence` monotonic counter (§5) continues across the rotation. An actor's full lifetime of signed events forms a single chain per frontier regardless of how many keys they have used.

7.6 Access Tiers

The access-tier model classifies evidence and event content into Public, Restricted, and Classified tiers. The state transition itself — the existence of an event, its kind, target, actor, timestamps, and dependency movement — is always inspectable; only the underlying evidence content may be tiered.

- **Public**: full content visible to anyone reading the corridor's state.
- **Restricted**: content held in trusted-reviewer escrow; only credentialed reviewers under the corridor's policy see the underlying evidence. The state transition itself is public.
- **Classified**: content held in regulator escrow; only the named regulator counterpart sees the underlying evidence. The state transition is public, signature-verifiable, and inspectable for chain-of-custody.

7.7 The Agent-Attestation Tier

Agent operators are a first-class actor category. An agent actor record extends the schema with three additional fields: a `tier` field set to `agent` (distinguishing it from `human` or `institution` tiers at schema level, not by string convention), an `operator` field carrying the stable identifier of the accountable human or organization, and a `stack_manifest` field carrying a content-addressed reference to the agent's stack description.

Agent deposits use a distinct event kind, `agent_attestation.deposited`, separate from `finding.asserted`. The semantic difference is canonical-merge authority: an `agent_attestation.deposited` event enters a pending-review state regardless of the agent's reputation; canonical merge into `finding.asserted` requires a human or institutional signature under §7.3 thresholds. The deposit is durable and citable as an attestation but does not, on its own, move canonical state.

A submission stake is required for agent deposits and forfeited on rule-based rejection. The stake is denominated in the agent's reputation graph (acceptance rate, calibration score) rather than in currency. A new operator with no reputation cannot bootstrap on stake alone — uninvited registration requires at least two credentialed sponsoring reviewers in the corridor (each with their own non-trivial reputation history and no shared institutional affiliation with the new operator). Sponsors stake their own reputation against the new operator's first 100 deposits with asymmetric incentive: liability proportional to rejection volume, gain proportional to acceptance-weighted-by-calibration quality (specifically, 15% of the sponsored operator's calibration-weighted contribution score during the probationary window accrues to the sponsor).

Multi-agent deposit pipelines — a retrieval agent feeding an extraction LLM feeding a tool-use verifier feeding a human reviewer — collapse to one accountable operator and one composite stack manifest at the canonical surface, but the protocol preserves the pipeline structure through trajectory steps. The trajectory primitive carries two structurally distinct

event kinds: `trajectory.step_appended` for scientific search-path steps (the original semantics), and `trajectory.pipeline_step` for agent-handoff steps (pipeline-provenance semantics).

Agent role types — proposer, reviewer, bridge-detector — carry an additional `role` field. Role-specific stake levels, sponsorship requirements, and merge-authority rules are defined by the corridor's Registry Governance Policy.

A reference RegistryGovernancePolicy template ships with v0.500 covering default sub-tier stake levels, sponsorship thresholds, and merge-authority rules. The template is a default, not a normative spec; corridors may fork it and tune values, but the template ensures early corridors do not have to invent these parameters from scratch.

7.8 The Agent Read Surface

The protocol provides a read surface specifically designed for agent consumption. The surface is defined in `crates/vela-protocol/src/agent_read.rs` and ships with v0.500+. SDK bindings for Python and TypeScript ship as part of the v0.500 milestone.

The read surface exposes: frontier state (the current view of findings, dependencies, confidence), event-log queries (recent activity, filtered by kind/actor/target), proof-packet retrieval (sealed state at named timestamps), and conformance metadata (which Carina kernel version is pinned, which reducer version produced the current state). Agents query the read surface under their actor record's access tier; classified content is filtered before return.

8 Governance Mechanics

8.1 Transparency-Log Witness

Governance events — kernel-version increments, witness-set changes, steward-tier policy updates, and ownership rotations — require cosignature by an independent transparency-log witness set under the regime adapted from Sigstore Rekor v2 and Sigsum.

A witness operates an append-only log of cosigned governance events. Witnesses do not produce canonical state; they attest that a governance event was published and cosigned at a specific time, and that the cosignature was witnessed by an independent party. Witnesses are independently operated by universities, foundations, scientific societies, and standards bodies (see `docs/WITNESSES.md` for the candidate set).

A governance event with `kernel_digest` field bound to a kernel release must be cosigned by at least M of N witnesses in the active witness set (M defined per witness-set version per §8.2) before any reducer accepts the event under strict mode. Witnesses have a defined freshness window: a cosignature older than 30 days from the event's timestamp is stale and triggers re-witnessing.

8.2 Witness-Set Governance

The witness set itself evolves under its own governance process. A witness-set version transition is itself a governance event carrying:

- The prior witness-set version
- The new witness-set version number
- The set of witnesses added (with their published keys)

- The set of witnesses removed (with revocation reasons)
- The cosigning requirement (M of N) for events under the new set

Witness-set transitions require cosignature by both the prior witness set and the new witness set. Historical events under prior witness sets remain valid forever; the `witness_set_version` field on each governance event (§5) lets the reducer accept the historical cosignatures without requiring the historical witness set to still be operating.

The protocol publishes witness-set versions, candidate witnesses, and cosigning statistics as canonical state surfaces. Witness governance is itself inspectable.

8.3 Kernel-Retraction Semantics

Carina is versioned. Each kernel version receives a content-addressed digest published to the transparency log. If a defect in a published kernel version is discovered after deployment — a signature-verification rule defect that allowed forged signatures to verify, a canonicalization rule that produced non-deterministic output across implementations, a reducer arm that mutated state incorrectly — the protocol records a `kernel.retraction_proposed` governance event.

If a retraction is severe enough that historical events under the retracted kernel must be considered unsound, the corridor's maintainer quorum may file a `frontier.kernel_remediation` event proposing migration of the frontier's history to a corrected kernel version. The migration is itself a series of governance-reviewed events: each historical event is re-validated under the new kernel, signatures are re-verified, and any event that fails re-validation is marked as quarantined pending review. Historical events that pass re-validation retain their original timestamps and signers; quarantined events are inspectable but excluded from canonical state. This is the protocol's recourse against fundamental kernel defects discovered late, and it explicitly trades off some replay-determinism guarantees for the ability to correct a load-bearing flaw without abandoning a corridor's accumulated history.

8.4 Federation through Peer Hubs

A peer hub mirrors a frontier's event log, validates signatures against published actor records, replays the log to materialize state, and exchanges new events with other peers. Peers do not rewrite history; they mirror, observe, and propose.

Federation operates at frontier granularity. A peer subscribes to specific frontiers by their `vfr_*` identifier and replays the genesis event forward. Conflict detection runs when a peer observes two events claiming the same `parent_event_id`; conflict resolution (§8.5) is a corridor-policy decision.

The current federation regime is point-to-point through peer hubs declared in each frontier's manifest; a canonical Registry layer is reserved for a future protocol version.

8.5 Conflict Detection and Resolution

When a peer observes two events claiming the same `parent_event_id`, the conflict is recorded as a `frontier.conflict_detected` event. The conflict is resolved through the corridor's Registry Governance Policy: the policy specifies which signer roles arbitrate, what evidence is required, and whether resolution requires a specific quorum.

Resolution itself is a governance event (`frontier.conflict_resolved`) carrying the winning event, the rejected event, and the reviewer attestation. Both the original conflicting events and the resolution event remain in the log; the resolution determines which event applies to canonical state going forward.

Disagreement that cannot be resolved within a single frontier may produce a fork: the dissenting reviewer signs a new genesis event for a forked frontier and the corridor's substrate becomes a graph of related frontiers rather than a single chain. Forkability is a constitutional commitment (§8.7).

8.6 Cross-Frontier References

A finding in one frontier may depend on a finding in another. Cross-frontier references carry the target frontier's `vfr_*` identifier, the target finding's `vf_*` identifier, and a pinned snapshot hash of the target frontier's state at reference time.

Under strict mode, a cross-frontier reference without a pinned snapshot hash is rejected. The pinned hash ensures that if the target frontier's state changes after the reference is made, the referring finding's dependency context is preserved. Implementations may operate under permissive mode (accepting unpinned references) for development; production corridors run strict.

8.7 Forkability

Any frontier or layer may be forked at any time. The fork inherits the full event log, signer graph, and dependency chain rather than starting from blank state. A fork is signaled by the forker filing a `frontier.forked` event in the new frontier's genesis chain, naming the source frontier and the divergence point.

The protocol guarantees that history travels with the fork. A bad-faith fork that omits history is structurally rejected: signature chains break, content addressing fails, and downstream peers refuse to mirror.

Forkability is a constitutional commitment, not an emergency mechanism. The expectation is that corridors will fork along legitimate disagreement — different reviewer pools, different governance policies, different access-tier defaults — and that forks will compose back when the disagreement resolves.

9 Cross-Implementation Conformance

§5 of the whitepaper claims byte-identical state across independent reducer implementations as the protocol's federation primitive. This section specifies the conformance surface and the path to closing the gaps.

Test vector set. The Vela repository ships a test vector suite at `crates/vela-protocol/tests/vectors/` containing genesis event sequences, expected after-hashes per event, and final state digests per frontier. Each vector is content-addressed and reproducible. The suite covers the 26 reducer mutation kinds documented in §6, with at least three vectors per kind exercising representative payload shapes and boundary conditions.

Conformance reporting. Each reducer implementation publishes a `conformance.json` artifact reporting pass-rate per kind per vector. The artifact is itself a signed deposit under the Vela

project's institutional actor record, so the conformance record is verifiable against the same protocol the implementations participate in. Published rates as of 2026-05-26 (Vela v0.500): Rust 100% (reference), Python approximately 78% (gaps in v0.55+ trajectory and evidence-atom materializers, ten kinds not yet supported), TypeScript approximately 41% (subset coverage against fixture data, twenty kinds not yet supported). Current rates are tracked at `crates/vela-protocol/conformance/` and supersede the snapshot quoted here.

Gating threshold for first-corridor deployment. The pilot will not begin recording canonical state for the BBB corridor until Python and TypeScript reducers each reach 95% pass rate against the v0.6 kernel vector set, and until at least two independent implementations (Rust + one of Python/TypeScript) sign the corridor's first proof packet without state-digest disagreement.

Ongoing conformance. Carina kernel version increments require all reducer implementations to publish updated conformance reports within 90 days of the increment, or the implementation is marked stale in the public conformance index. Implementations that fall stale for more than two consecutive kernel versions are removed from the federated peer-discovery list at corridor governance discretion.

This conformance regime is more rigorous than what most protocol specifications commit to, and it is achievable because the protocol is content-addressed and deterministic at the reducer interface. The Vela project commits to maintaining the test vector set as a load-bearing public artifact, on equal footing with the protocol specification itself.

10 Canonical JSON Specification

Replay determinism (§4) and signature verification (§7.2) both depend on every implementation producing byte-identical canonicalized JSON for the same input. Without an explicit canonicalization specification, divergence between implementations is inevitable — and the Python (~78%) and TypeScript (~41%) reducer parity gaps reported in §9 for Vela v0.500 as of 2026-05-26 trace primarily to incomplete canonicalization conformance.

The protocol adopts **RFC 8785 (JSON Canonicalization Scheme, JCS)** as its canonicalization standard, with the additions noted below for protocol-specific values. JCS specifies a deterministic serialization for JSON values such that any two implementations that follow the standard produce the same byte sequence for the same value. The relevant rules:

Key ordering. Object keys are sorted lexicographically by their UTF-16 code unit sequence (RFC 8785 §3.2.3). Nested objects sort independently at each level. Arrays preserve their original order; only object keys are sorted.

String encoding. Strings are serialized in UTF-8, NFC-normalized (Unicode Standard Annex #15). Control characters U+0000 through U+001F are escaped as `\u00XX`; the characters `"` and `\` are escaped as `\"` and `\\`; characters U+007F through U+009F are escaped as `\u00XX`. All other characters are emitted as raw UTF-8. The forward-slash character is not escaped.

Number serialization. Numbers follow JCS's IEEE 754 double-precision rules. Integers (values exactly representable as integers within the safe-integer range −2^53+1 to 2^53-1) are serialized without a decimal point and without an exponent. Non-integer values use the shortest decimal representation that round-trips to the same double-precision value (ECMAScript 2017 Section 6.1.6.1). Negative zero serializes as `0` (the sign is dropped). Infinity and NaN are not

representable in canonical JSON; an attempt to canonicalize a value containing them is an implementation error.

Boolean and null. Serialized as `true`, `false`, and `null` respectively (lowercase, no whitespace).

Absent vs. null fields. The protocol treats absent and null fields as semantically distinct. A field present with value `null` is canonicalized as `"key":null`; the same field omitted from the object is canonicalized as if the key did not exist. Implementations must not silently elide null fields. Schema migrations that change a field from required to optional must explicitly specify whether existing serializations should be re-canonicalized.

Whitespace. No whitespace appears between tokens in canonicalized output. No leading or trailing whitespace surrounds the serialization.

Order of operations. A protocol-defined value is canonicalized in the following order: (1) the value is recursively walked and normalized (strings NFC, numbers to canonical form, object keys sorted); (2) the normalized value is serialized to UTF-8 bytes per the rules above; (3) signatures over the canonicalized value cover the byte sequence directly, with no additional padding or framing.

Signature self-reference rule. When canonicalizing a payload for signature production or verification, the `signatures` field is treated as absent — present-with-empty-array and absent are equivalent at canonicalization time. This breaks the circular dependency that would otherwise arise from a struct that carries the signatures over its own content. Verifiers reconstruct the canonicalized payload by stripping the `signatures` field, recomputing the canonical bytes, and verifying each signature against the resulting byte sequence.

Protocol-specific values. Several Vela primitives extend beyond RFC 8785's specification:

- `Hash` strings (sha256 digests, kernel digests, content hashes) are canonicalized as `sha256:` followed by the 64-character lowercase hexadecimal representation. Mixed-case or uppercase hex is invalid.
- `Identifier` strings (`vf_*`, `vev_*`, `vfr_*`, etc.) are case-sensitive and treated as opaque strings; the protocol does not normalize case in identifiers.
- `Timestamp` strings follow RFC 3339 with mandatory UTC offset (`Z` or `+00:00`); local-time offsets are converted to UTC before canonicalization. Sub-second precision is preserved verbatim if present.
- `Signature` byte sequences are encoded as lowercase hexadecimal in the canonical representation; canonicalization computes the digest over the underlying byte sequence, not over the hex string.

Conformance. Implementations claiming Vela protocol conformance must pass the canonicalization test vector suite at `crates/vela-protocol/tests/vectors/canonicalization/`, which exercises edge cases including: NFC normalization of composed and decomposed Unicode forms, integer-vs-float boundary serialization, absent-vs-null field handling, deep nesting, large arrays, and high-codepoint string content. The 95% conformance threshold for first-corridor deployment (§9) is measured against this suite.

Reference. Implementers should treat RFC 8785 as the normative document; this specification specifies only the extensions and clarifications relevant to Vela. The protocol's Rust reference implementation lives in `crates/vela-protocol/src/canonical.rs`.

11 Lean Soundness Theorem

The governance soundness theorems for Vela are mechanized in Lean at `lean/Vela/`. The load-bearing result is Theorem 16 (verify_quorum soundness), at `lean/Vela/GovernedQuorumSoundness.lean`:

> If a `verify_quorum` call returns `true` for an event under a Registry Governance Policy (`vgp_*`), then the set of signatures presented satisfies the policy's quorum requirement, no signer appears more than once, every signer is in the eligible-actors set at the event's timestamp, no signer's actor record carries a `revoked_at` timestamp before the event's timestamp, and the role constraints (if present) are met.

The theorem's statement is byte-identical between the Lean source and the prose in the whitepaper §6 (Identity and Signing). Future revisions will mechanize additional soundness properties — reducer determinism, conflict-resolution termination, kernel-retraction safety — as they reach formal-proof maturity.

12 Test Vector Suite Reference

The test vector suite at `crates/vela-protocol/tests/vectors/` is structured as one subdirectory per mutation kind. Each subdirectory contains:

- `genesis.json` — the genesis event for a synthetic frontier
- `events.json` — the ordered sequence of events to replay
- `expected-after-hashes.json` — the expected `after_hash` for each event in sequence
- `expected-state-digest.json` — the expected finding-state digest at end of replay
- `notes.md` — the vector's purpose and any edge cases it exercises

The canonicalization suite at `crates/vela-protocol/tests/vectors/canonicalization/` is structured by edge-case category: `nfc-normalization/`, `integer-float-boundary/`, `absent-vs-null/`, `deep-nesting/`, `large-arrays/`, `high-codepoint-strings/`. Each subdirectory contains input JSON, expected canonical bytes, and reference notes.

Implementations report conformance per-kind and per-canonicalization-category. The published rate is the weighted average across all categories.