Proof of Insight v0.7.0

The Proof of Insight™ Protocol

Version 0.7.0 — Working Draft

Status. This document is a working draft circulated for review by domain experts in cryptographic provenance, regulated artificial intelligence, and standards development. It is not a published standard. References to companion documents (conformance test suite, reference verifier, profile bindings, rendering specification, reviewer's guide) are forward references; those documents do not yet exist at the version cited.

Versioning. The value `"0.7.0"` appears in the `version` field of every Insight Step (§2.1), the `manifest_version` field of every proof manifest (§2.7), and the `version` field of every manifest-level attestation (§2.9). Document and protocol version are intentionally the same.

Version 0.7.0 supersedes v0.6.2 with a set of changes that includes breaking changes to the canonical encoding and step-identity rule. v0.6.2 proofs are not valid v0.7.0 proofs and must be re-issued, not re-labeled. This is the first — and is intended to be the only — pre-1.0 revision that alters identity computation. The rationale for taking these changes in a single minor-version step, rather than as a sequence of patch releases, is given in Appendix E. The protocol remains pre-1.0 and no production deployments exist at any version; the cost of this break is borne now, deliberately, while that cost is zero.

The substantive changes are summarized in Appendix D: (1) step identity is computed over the signed step content *excluding* the timestamp, with the timestamp authority attesting to the step identity directly; (2) all digests and signatures are carried as structured objects with explicit algorithm identifiers; (3) canonical encoding of non-JSON artifacts is defined, closing a normative gap; (4) a coverage mechanism over prespecified analysis inventories makes selective recording of analyses mechanically detectable; (5) disclosure-limited (redacted) artifacts are given defined dual-commitment semantics; (6) archival proof bundles and a verification report format are specified; (7) manifest-level attestations, deferred from v0.6.2, are specified; (8) the L4A independence requirement is strengthened from ″distinct identity″ to a graded independence predicate.

Editor's note. This draft prioritizes precision in §1–§3 and §5 (the technical core and conformance machinery) over completeness in §4, §6–§8. Section §6 remains an enumerated placeholder; Appendix A's machine-readable schema is to be regenerated against the v0.7.0 field set (the required deltas are enumerated in the Appendix A editor's note).

---

§0. Copyright, License, and IPR

§0.1 Document. This specification is © 2026 Arclio LLC and is licensed under the Creative Commons Attribution 4.0 International License (CC BY 4.0). The full license text is available at https://creativecommons.org/licenses/by/4.0/legalcode. The canonical version of this document is published at https://proofofinsight.org/spec/v0.7.0/.

§0.2 Reference artifacts. The JSON schema of Appendix A is illustrative and is licensed under CC BY 4.0 with the rest of this document. Normative machine-readable schemas, the reference verifier implementa-

tion, and the conformance test suite are published separately under the Apache License, Version 2.0, including its patent grant provisions.

§0.3 Patent non-assertion. Arclio LLC covenants not to assert any patent claims it owns or controls against any implementation of this specification that conforms to the conformance test suite at the version of this document under which the implementation claims conformance. Until a conformance test suite is published for a given version of this specification, the covenant applies to implementations that make a good-faith implementation of the normative requirements of §§1–5 at that version. This covenant binds Arclio LLC only; it does not address claims that may be held by third parties.

§0.4 Trademarks. ″Proof of Insight″ and ″PoI″ are trademarks of Arclio LLC. Use of these marks to describe an implementation as conformant is governed by the PoI Conformance Mark Policy, published separately. Use of the marks in this specification and in any derivative work is permitted for the purpose of accurate reference to this specification.

§0.5 Editorial process. This specification is a working draft. Editorial control rests with Arclio LLC pending establishment of an independent editorial body. Proposed changes are submitted via the public issue tracker referenced at the canonical URL.

Abstract

This document specifies the Proof of Insight™ (PoI™) protocol, an evidence format for analyses produced by agentic artificial-intelligence systems in regulated contexts. PoI defines a content-addressed, signed directed acyclic graph (DAG) of typed derivation steps. A single verification algorithm, parameterized by step type, establishes that recorded outputs are *bound* to recorded inputs through recorded operations and attestations, that the binding is tamper-evident, that deterministic operations are replayable, and that non-deterministic operations are recorded under stated conditions with explicit re-execution semantics. Where the producer prespecifies an inventory of analyses, the protocol additionally establishes that every prespecified analysis is accounted for in the proof — by a conclusion, a no-finding result, or an explicit supersession — making selective recording mechanically detectable (§5.6).

PoI extends existing software-supply-chain attestation frameworks (in-toto, SLSA) and provenance data models (PROV-W3C) by admitting non-deterministic reasoning operations as first-class evidence steps and by providing a single mechanical verification algorithm over the resulting graph. As with those frameworks, PoI's verification claims concern the *provenance and integrity* of an analysis, not the *semantic correctness* of its conclusions; this distinction is constitutive of the protocol's scope and is stated explicitly in §1.3 and §1.4. The protocol is independent of any particular implementation, signing scheme, timestamp authority, or compliance regime; bindings to specific ecosystems are provided as informative profiles. A self-contained archival bundle format (§2.8) and a verification report format (§3.5) are specified so that proofs can be verified, and verification outcomes recorded as evidence, without network access to producer-operated services.

---

1. Introduction

1.1 Problem

Agentic artificial-intelligence systems are increasingly used to produce analyses that inform regulated decisions. Examples include readiness assessments for drug-development submissions, model-risk reviews in finance, decision support in clinical care, and reporting under environmental-compliance regimes. Such systems characteristically combine three classes of operation:

1. Ingestion of external data (clinical trial records, market data, sensor logs).
2. Deterministic computation over that data (database queries, statistical primitives, deterministic transformations).

3. Non-deterministic reasoning, typically realized by large language model inference, which interprets, summarizes, or draws conclusions from the results of the previous two classes.

A substantial body of prior work addresses the first two classes. PROV-W3C provides a general data model for provenance. The in-toto and SLSA frameworks provide verifiable attestation for software supply chains, with Sigstore providing a deployed signing and transparency-log infrastructure. These frameworks were designed before agentic reasoning was a routine component of regulated workflows and they assume the operations they record are deterministic builds. None admits non-deterministic reasoning as a first-class operation; none provides a verification algorithm of the precision a regulator requires to act independently of the producer over a graph that includes such operations.

PoI builds on this lineage rather than replacing it. PoI's signature, content-addressing, and timestamping primitives are intended to be instantiated via the existing ecosystems (see profiles in §7). PoI's contribution is the typed-step taxonomy that admits non-deterministic reasoning, the typed edge relations that distinguish input from context from attestation, and the single mechanical verification algorithm parameterized by step type.

The motivating consequence is that analyses produced by agentic systems in regulated contexts currently rely on producer-controlled audit logs, narrative descriptions, or post-hoc reconstructions. None of these is mechanically verifiable. A regulator presented with such an analysis has no protocol-level basis on which to confirm even the structural integrity of the analysis — that the recorded outputs are bound to the recorded inputs through the recorded operations, without tampering, under identifiable attestation. PoI provides that structural basis. It does not, and no provenance protocol can, certify that the recorded analytical conclusions are *correct*; see §1.4.

A second motivating consequence, addressed for the first time in this version, is selective recording: a proof attests to what it contains, never to what was run. An analysis that produced an inconvenient result can simply be omitted. Where the producer has prespecified what will be analyzed, v0.7.0 makes that omission mechanically detectable (§5.6). Where nothing was prespecified, the protocol remains — necessarily — silent; no evidence format can witness work that was never recorded.

1.2 Position

PoI is a verification protocol, not an analytical platform or workflow tool. Platforms that produce analyses — whether for clinical, financial, or other regulated contexts — may emit PoI proofs of those outputs to support independent verification. The protocol's design does not depend on, nor does it specify, the workflow that produced an analysis; it specifies the evidence structure against which that analysis can be mechanically verified.

PoI's central position is that a single primitive — a content-addressed, signed, typed derivation step — is sufficient to express the evidence required for regulated agentic analyses, and that the properties such evidence must possess emerge as theorems over composed steps rather than being designed in as separate features.

A PoI proof is a directed acyclic graph of such steps. Verification is a single algorithm parameterized by step type. Conformance levels are defined as constraints on which step types must appear and which roles must sign which steps.

This position commits the protocol to a small, fixed taxonomy: four step types (`observe`, `compute`, `reason`, `attest`) and three edge relations (`derived-from`, `conditioned-on`, `about`). Domain-specific specialization — clinical prespecification, financial model tiering, sector-specific roles — is achieved through the profile mechanism (§7.0) without modifying the base taxonomy. The taxonomy is justified in §2. The v0.7.0 additions (coverage, redaction, bundles, reports, manifest-level attestations) are deliberately built *from* this taxonomy and the manifest, not as new step types; the four-type taxonomy is unchanged.

1.3 Non-goals

PoI explicitly does not provide:

- **Validation of input data.** PoI binds an analysis to its claimed inputs through content hashing and attestor signature. It does not certify that the inputs are accurate, complete, current, or fit for purpose. Input validation is the responsibility of the attestor and any external authorities to which the attestor delegates (e.g., source-data integrity standards, vendor-qualification regimes).
- **Validation of model outputs.** PoI records that a model produced a specific output under recorded conditions. It does not certify that the output is correct, calibrated, or safe.
- **Endorsement of analytical conclusions.** PoI's verification algorithm establishes that recorded outputs are bound to recorded inputs through recorded operations and attestations, with the binding tamper-evident. For deterministic operations, replay can additionally confirm that the recorded output is the one the recorded function produces on the recorded inputs. For non-deterministic operations, the protocol records the operation's conditions and output but does not establish that the output follows from the inputs in any logical, statistical, or regulatory sense. Whether any recorded conclusion is correct, defensible, or appropriate remains a matter for human review and regulatory judgment, outside the scope of the protocol.
- **Completeness of unprespecified work.** The coverage mechanism (§5.6) establishes that every analysis enumerated in a prespecified inventory is accounted for in the proof. It does not, and cannot, establish that analyses *outside* any inventory were not run and discarded. Coverage converts selective recording from undetectable to detectable exactly to the extent that the producing organization prespecifies its work.
- **A consensus mechanism.** PoI proofs are produced by a single producer or a coordinated set of producers under a shared trust model. Distributed agreement among independent parties on the contents of a proof is out of scope.
- **A retention or archival policy.** PoI specifies how evidence is structured, packaged (§2.8), and verified. It does not specify how long evidence is kept, by whom, or under what conditions it may be deleted. The archival bundle of §2.8 is a *packaging* format that makes long-horizon verification possible; the obligation to retain bundles, and for how long, belongs to the applicable records-retention regime.
- **A trust root for identity.** PoI requires that every step be signed by an identifiable attestor, but does not specify how identities are issued, verified, or revoked. Identity is a profile concern (§7). v0.7.0 adds one base-protocol requirement at this boundary: authorization is evaluated as of signing time, not verification time (§3.2), which obligates profiles to support historical authority resolution.

These non-goals are constitutive of the protocol's scope. A reader who expects PoI to provide any of the above is reading the wrong document. The companion scope statement in §1.4 makes the verification-versus-credibility distinction explicit.

1.4 Scope: Verification, Not Credibility

PoI establishes that an analysis was produced as recorded. It is silent on whether the recorded analysis was the right analysis. This distinction is consequential and is stated here as part of the protocol's primary scope, not as a footnote.

A proof that satisfies all conformance checks of §5 establishes:

- That the recorded inputs are bound to the recorded operations through identified attestations
- That the binding is tamper-evident
- That deterministic operations are replayable under their declared regime
- That non-deterministic operations are recorded with stated conditions and explicit re-execution semantics
- That predicates over the typed-step structure (e.g., "every reason step on the path to a regulated conclusion has at least one qualified-reviewer attestation") hold mechanically
- Where a prespecified analysis inventory is present: that every enumerated analysis maps to a recorded outcome — positive, negative, or no-finding — or to an explicit supersession (§5.6)

A proof does not establish:

- That the recorded inputs are accurate, complete, or fit for purpose
- That the recorded methods are appropriate to the question being asked

- That the recorded conclusions are correct, calibrated, or defensible
- That a reviewer applying domain judgment would reach the same conclusion
- That the analysis is sufficient to support any specific regulatory action
- That work outside the recorded proof — including analyses never enumerated in any inventory — did not occur

The protocol therefore answers a *process-fidelity* question and is silent on the *analytical-correctness* question. Both questions matter for regulated decisions, but they are addressed by different mechanisms: PoI by mechanical verification, analytical correctness by qualified human review under domain-specific standards. PoI is structured so that an `attest` step from a qualified reviewer (§2.2.4) records the latter and binds it to the former. The attest step records the review; it does not perform it.

Reproducibility of an analysis under PoI is not equivalent to credibility of that analysis. A reproducibly wrong analysis remains wrong. The protocol's value to a regulator is that it makes the analysis legible and mechanically inspectable, so that human judgment can focus on the questions that require it.

1.5 Conventions

The key words MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, NOT RECOMMENDED, MAY, and OPTIONAL in this document are to be interpreted as described in BCP 14 (RFC 2119, RFC 8174) when, and only when, they appear in capitals.

Digest objects. Wherever this document carries a digest, it is carried as a structured *digest object*:

```
digest = {
  "alg":   <lowercase algorithm identifier, e.g. "sha-256">,
  "value": <lowercase hex encoding of the digest bytes>
}
```

The base protocol registers a single algorithm identifier, `"sha-256"`, denoting SHA-256. Profiles MAY register additional identifiers (including post-quantum algorithms); the registry mechanism is the `alg` field itself, and canonical encoding rules accommodate substitution without changes to the protocol. Unqualified references to ″hash,″ ″digest,″ or ″SHA-256″ in this document denote a digest object with `alg` `"sha-256"` unless a profile specifies otherwise. Step identities (§2.5) are digest objects.

Signature objects. Wherever this document carries a signature, it is carried as a structured *signature object*:

```
signature = {
  "alg":    <signature scheme identifier; profile-registered>,
  "key_id": <optional identifier of the signing key within the attestor's key set>,
  "value":  <base64 encoding of the signature bytes>
}
```

The base protocol does not register signature schemes; schemes are profile-defined. The structured form exists so that scheme metadata travels with the signature rather than being implied by out-of-band profile knowledge.

Canonical encoding of JSON values follows RFC 8785 (JSON Canonicalization Scheme) unless a profile specifies an alternative. Canonical encoding of artifacts that are not JSON values is specified in §2.5.1.

The term ″attestor″ denotes the identified party signing a step. The term ″verifier″ denotes any party — including but not limited to the producer of the proof — running the verification algorithm of §3 against a proof. The term ″producer″ denotes the system or organization assembling a proof from steps it generates. The term ″authorized verifier″ denotes a verifier granted access, under a profile's access model, to disclosure-limited artifacts (§2.2.5).

---

2. The Insight Step

A PoI proof is composed of *insight steps* (hereafter "steps"). This section specifies the step structure, the four step types and their payloads, disclosure-limited artifacts, the three edge relations, the timestamp binding, the canonical encoding and identity rule, the construction-time invariants, the proof manifest, the archival bundle, and manifest-level attestations.

2.1 Step structure

Every step is a record with the following fields. Field order is fixed by the canonical encoding rule of §2.5; the order in this table is the canonical order.

#	Field	Type	Required	Description
1	`version`	string	MUST	Protocol version. This document specifies `"0.7.0"`.
2	`type`	string	MUST	One of `"observe"`, `"compute"`, `"reason"`, `"attest"`.
3	`predecessors`	array of edges	MUST	Edges to predecessor steps. MAY be empty only for `observe`.
4	`payload`	object	MUST	Type-specific. Contents specified in §2.2.
5	`attestor`	string	MUST	URI identifying the attestor. Resolution is profile-defined.
6	`signature`	signature object	MUST	Signature over the canonical encoding of fields 1–5, by the key bound to
7	`timestamp`	object	MUST	Trusted timestamp binding to the step's identity (§2.4, §2.5). Carried witl

The ordering of signature *before* timestamp is deliberate. The attestor signs the unsigned step content; a timestamp authority then attests that the *signed* step existed at a stated time. This binding order — sign, then timestamp the signed thing — establishes that the signed evidence existed at the time the timestamp authority attests, not merely that the unsigned content existed.

For canonicalization, two byte strings are defined:

```
to_sign(step)      = canonical_encoding( fields 1..5 of step )
to_timestamp(step) = canonical_encoding( fields 1..6 of step )   // includes signature
```

The attestor signs `to_sign(step)` to produce `signature`. A step's *identity* (§2.5) is the digest of `to_timestamp(step)`. The timestamp authority attests to the existence of the step's identity — and therefore, by preimage, of the signed step content — at the timestamp value, producing `timestamp.token` (§2.4).

Change from v0.6.2 (design note). In v0.6.2 the identity was computed over all seven fields, including the timestamp token. v0.7.0 computes identity over fields 1–6 and excludes the timestamp. Three consequences motivated the change. First, *construction latency*: because successors reference predecessor identities, the v0.6.2 rule serialized proof construction on timestamp-authority round-trips — step *n+1* could not be encoded until step *n*'s token returned. Agentic runs producing hundreds of steps make this a real cost. Second, *token nondeterminism*: timestamp tokens are opaque authority-produced blobs; identical content timestamped twice yields different tokens and therefore, under v0.6.2, different identities. Third, *authority migration*: under v0.6.2, a step whose timestamp authority later ceased to be profile-recognized could not be re-timestamped without changing its identity and invalidating every descendant.

The cost of the change is that a specific timestamp token is no longer pinned into descendants' hashes. This cost is bounded: the token is self-authenticating over the step identity, so a timestamp cannot be *altered* undetectably (token verification fails); it can only be *replaced* by another genuine token from a recognized authority over the same identity. Replacement by a later genuine token only moves the evidence later in time, which never strengthens a producer's position (prespecification and ordering checks favor earlier timestamps). Replacement by an earlier fraudulent token requires authority compromise, which is threat 7 in §6 and was equally fatal under v0.6.2. Profiles requiring global uniqueness or monotonicity of timestamps SHOULD bind to transparency-log authorities (e.g., Rekor inclusion proofs), under which every issued timestamp is publicly enumerable.

A step is verified in three layers, in order: signature validity over `to_sign`; identity computation over `to_timestamp`; timestamp validity over the identity (§2.4). A verifier MUST perform all three.

2.2 Step types and payloads

PoI defines exactly four step types. The taxonomy is minimal in the sense that no two types may be collapsed without losing a verification distinction that this protocol relies on; it is not extensible without a protocol revision. Domain-specific predicates that might naively appear to require additional step types are instead expressed via the `attest` step's claim-type vocabulary (§2.2.4) under a profile (§7.0).

2.2.1 **`observe`**

An `observe` step records the ingestion of external data into the proof.

```
payload = {
  "content_hash":  <digest of the observed artifact's canonical encoding (§2.5.1)>,
  "content_type":  <string: MIME type or domain identifier>,
  "source":        <URI or structured descriptor of origin>,
  "provenance":    <optional reference to external provenance attestation>
}
```

An `observe` step MUST NOT have predecessors. It is the only step type for which `predecessors` MAY be the empty array.

A verifier cannot independently re-derive an `observe` step. Verification of an `observe` step is limited to: (a) confirming, where the original artifact is separately available, that its canonical encoding digests to `content_hash`; (b) confirming that the attestor was authorized — under the profile's authority model, as of the step's timestamp (§3.2) — to make observations from the named source; (c) verifying any external provenance attestation referenced in `provenance`.

The `provenance` field is the integration point with external attestation ecosystems (e.g., in-toto attestations describing the origin of input data). Such external attestations are not part of the PoI proof but are referenced from it.

The observe step as trust handoff boundary. Data exists in the analysis as PoI evidence from the moment of the signed `observe` step; what occurred upstream — including how the data was collected, validated, qualified, or amended — is the responsibility of source-data integrity standards (e.g., 21 CFR Part 11 §11.10 for clinical data, books-and-records rules for financial data) and any external provenance attestations referenced via the `provenance` field. A verifier confirming an `observe` step confirms that the recorded content digests to the recorded value and that the attestor was authorized to make observations from the named source; it does not confirm that the source itself is trustworthy. This boundary is intentional and is the reason source-data validation is enumerated as a non-goal in §1.3.

Observe timestamps record ingestion, not observability. An `observe` step's timestamp establishes when the artifact entered the proof, not when the data became observable to any analyst. Mechanisms that depend on data-exposure ordering — prespecification above all — MUST NOT treat observe timestamps as evidence of first exposure. See §5.1 (L4A requirement 2) and the clinical profile sketch (§7.4), which binds unblinding to external events rather than to observe timestamps.

2.2.2 **`compute`**

A `compute` step records the application of a deterministic function to inputs derived from predecessor steps.

```
payload = {
  "function":        <URI or content-digest identifier of the function>,
  "invocation":      <inline canonical invocation object | content-addressed
                      reference of form { "uri": <URI>, "digest": <digest> }>,
  "invocation_hash": <digest of canonical encoding of the invocation object>,
  "output_encoding": <"jcs+json" | "octet-stream" | profile-defined identifier>,
  "output_hash":     <digest of canonical output encoding (§2.5.1)>,
```

```
  "output_artifact": <optional artifact carrier (§2.2.5): inline, by reference,
                      or disclosure-limited; required for tolerance replay
                      regimes; see §3.2>,
  "environment":     <execution environment descriptor; see below>
}
```

The *canonical invocation object* — referenced inline or by content address in `payload.invocation` — is the structured record that fully determines the function's inputs:

```
invocation = {
  "function":   <same as payload.function>,
  "inputs": [
    {
      "name":        <local name used inside the function>,
      "step":        <identity digest of predecessor step>,
      "output_hash": <digest of that predecessor's output; redundant with the
                      predecessor record but recorded here for verifier
                      convenience and to bind the linkage>
    },
    ...
  ],
  "parameters": <function parameters not sourced from predecessor outputs>
}
```

`invocation_hash` is the digest of the canonical encoding (§2.5) of this invocation object and MUST equal the digest computed by the verifier over the resolved `payload.invocation` (whether inline or fetched from the content-addressed reference). Hashing the full invocation — inputs *and* parameters together — closes the linkage between predecessor outputs and the step that consumes them. Two compute steps with identical `parameters` but different predecessor bindings produce different `invocation_hash` values.

The verifier MUST be able to obtain the invocation object. If `payload.invocation` is inline, it is read directly. If it is a content-addressed reference, the verifier MUST resolve the reference (the resolution mechanism is profile-defined; within an archival bundle, resolution is against the bundle's artifact store, §2.8) and confirm the fetched object's canonical encoding digests to the reference's stated digest. Either form is acceptable; profiles MAY constrain which is permitted for their conformance level.

A `compute` step MUST have at least one predecessor edge, and every predecessor edge MUST have relation `derived-from`. A `compute` step MUST NOT have predecessor edges of any other relation. The set of `derived-from` predecessors MUST equal the set of step identities appearing in `invocation.inputs`. A `compute` step MUST NOT have any `derived-from` predecessor of type `attest` (see §3.0 on `STEP_RESOLVE`).

`output_encoding` declares which canonical artifact encoding rule (§2.5.1) governs `output_hash`. It is REQUIRED. This field exists because bit-identical replay is only well-defined when producer and verifier agree on the byte representation of the output; in v0.6.2 this agreement was implicit and unspecified.

The `environment` descriptor specifies the conditions for replay. It MUST include a `replay_regime` field with value either `"bit-identical"` or `"tolerance"`:

- **`bit-identical`**: the digest of the canonical encoding of the replayed output MUST equal `payload.output_hash`. `payload.output_artifact` MAY be omitted.
- **`tolerance`**: replay produces an output that must be equivalent to the recorded output under a profile-defined equivalence predicate. `payload.output_artifact` MUST be present (inline, by content-addressed reference, or disclosure-limited per §2.2.5 — noting that tolerance replay against a disclosure-limited artifact is available only to authorized verifiers), and the verifier compares *outputs* under the equivalence predicate, not digests. The `environment` MUST additionally specify the basis for the relaxed requirement (e.g., ″non-associative floating-point reductions across BLAS implementations″) and the equivalence predicate identifier the profile applies.

A `compute` step's verification semantics depend on whether the function and environment are resolvable to the verifier; details in §3.2.

2.2.3 **`reason`** A `reason` step records the application of a non-deterministic model — characteristically a large language model, but not exclusively so — to inputs derived from predecessor steps.

```
payload = {
  "model": {
    "identifier":   <URI or string identifier>,
    "weights_hash": <digest of weights, where available>,
    "version":      <string, where applicable>
  },
  "replay_class":   <"R1" | "R2" | "R3">,
  "invocation":            <inline canonical reason-invocation object |
                            content-addressed reference { "uri", "digest" }>,
  "invocation_hash":       <digest of canonical encoding of invocation>,
  "input_messages":        <artifact carrier (§2.2.5): inline canonical input
                            message sequence, content-addressed reference, or
                            disclosure-limited>,
  "input_messages_hash":   <digest of canonical encoding of input messages;
                            for disclosure-limited carriage this is the
                            binding digest of the unredacted sequence>,
  "tool_call_log":         <optional artifact carrier: the canonical sequence
                            of tool calls and responses occurring during
                            inference; absent for non-tool-calling steps>,
  "tool_call_log_hash":    <digest of canonical encoding of the tool-call
                            sequence, if any>,
  "visible_rationale":     <optional artifact carrier: any model-produced
                            explanation, rationale, reasoning summary, or
                            analytic narrative exposed to users or reviewers>,
  "visible_rationale_hash": <digest of canonical encoding of the visible
                             rationale; absent if not applicable>,
  "finding_type": <"conclusion" | "no-finding" | "insufficient-evidence"
                 | "negative-result" | <profile-defined>>,
  "output_encoding": <"jcs+json" | "octet-stream" | profile-defined>,
  "output_hash":     <digest of canonical output encoding>,
  "output_artifact": <optional artifact carrier; required when
                      replay_class = R1 (see below)>,
  "sampling": {
    "temperature":  <number>,
    "seed":         <integer | null>,
    "top_p":        <number, optional>,
    "top_k":        <integer, optional>,
    ...
  },
  "redactions": <optional redaction record (§2.2.5) enumerating which
                 artifact fields of this payload are disclosure-limited
                 and under which registered policy>
}
```

If `finding_type` is omitted, the value defaults to `"conclusion"`. The protocol's structural treatment of non-conclusion findings is specified in §5.5.

Change from v0.6.2. v0.6.2 carried `tool_call_log_hash` and `visible_rationale_hash` with no corresponding artifact carrier: the digests existed but the artifacts themselves had no defined home in the step, leaving their

retrievability entirely to out-of-band resolution. A digest without a retrievable preimage is weak evidence — it pins the producer to *something* without letting a reviewer see *what*. v0.7.0 adds the `tool_call_log` and `visible_rationale` carriers. When a hash field is present, the corresponding carrier SHOULD be present (inline, by reference, or disclosure-limited) whenever the proof is intended to be reviewable without replay; profiles MAY require it. The free-standing `redaction_policy` field of v0.6.2 is replaced by the structured `redactions` record of §2.2.5.

The *reason invocation object* — referenced inline or by content address in `payload.invocation` — is the structured record that determines the model's input:

```
reason_invocation = {
  "model":          <same as payload.model>,
  "input_bindings": [
    { "name": <name>, "step": <id digest>, "output_hash": <digest> },
    ...
  ],
  "input_messages_hash":  <same as payload.input_messages_hash>,
  "context_frame": {
    "conditioned_on": [ <step id digest>, ... ]
  },
  "sampling": <same as payload.sampling>
}
```

`invocation_hash` is the digest of the canonical encoding of this object. The `input_bindings` list corresponds to `derived-from` predecessors; the `context_frame.conditioned_on` list corresponds to `conditioned-on` predecessors (see below). As with `compute`, the verifier MUST be able to obtain both the invocation object and the input messages — where the input messages are disclosure-limited, ″obtain″ is satisfied by the disclosed form per §2.2.5 — and MUST confirm that each resolves to an object digesting to its declared digest (for disclosure-limited artifacts, to the *disclosed* digest; the binding digest is checkable only by authorized verifiers).

A `reason` step MUST have at least one predecessor edge. Predecessor edges MAY have relation `derived-from` (the predecessor's output is part of the model's input — typically bound by name into one of the input messages) or `conditioned-on` (the predecessor is declared as evidentiary context relevant to the current step but is not a direct input to the operation). A `reason` step MUST NOT have predecessor edges of relation `about`. A `reason` step MUST NOT have any `derived-from` predecessor of type `attest` (see §3.0).

The verifier reconstructs the input from `derived-from` predecessors and confirms the reconstruction digests to `input_messages_hash` (§3.2). The verifier does *not* mechanically verify how, or whether, `conditioned-on` predecessors affected the operation; their inclusion is an assertion by the attestor that the listed steps are evidentiary context for the current operation. This assertion is surfaced to human reviewers and to compliance predicates (§5), but is not subject to replay verification. The `conditioned-on` relation is therefore primarily a reviewability and regulatability mechanism, not a replay mechanism.

Visible rationale as evidence. The visible-rationale fields record any model-produced explanation, rationale, reasoning summary, or analytic narrative exposed to users or reviewers. Unlike the output fields (which record the model's conclusion) and the tool-call fields (which record mechanical tool interactions), the visible rationale records the model's *narrative* of its own analysis as exposed for review. Recording the rationale as content-addressed evidence — rather than as a soft ″explanation″ field — is one of the central mechanisms by which PoI extends provenance frameworks into agentic reasoning: the rationale is hash-bound to the inputs that produced it, the operation that produced it, and any downstream step that consumes it. A reviewer auditing a `reason` step inspects the rationale; a verifier confirms the rationale digests correctly; a downstream `attest` step can be made about the rationale specifically.

The protocol does not assert that the recorded rationale is a faithful representation of the model's hidden reasoning processes. Many systems do not expose their internal reasoning at all, and exposed text is often a post-hoc summary or generated explanation rather than a trace of the actual computation. The protocol

records what was exposed for review; it does not claim that what was exposed is the reasoning. The evidence value of the visible rationale is that it pins the producer to the explanation it surfaced, not that the explanation is mechanistically true.

Tool-call granularity. The `tool_call_log` records the full sequence of tool calls and their responses during inference as a single canonical artifact, digested as a unit by `tool_call_log_hash`. The default granularity is therefore the entire tool-call sequence. A profile MAY require expanded representation — for example, by mandating that each tool call be recorded as its own `compute` substep with explicit `derived-from` edges, with the `reason` step then taking the substeps as predecessors. Expanded representation produces a strictly larger and more inspectable proof at the cost of greater encoding overhead.

Profiles whose regulated scope includes `reason` steps that are ancestors of regulated conclusions SHOULD require expanded representation for those steps. The motivation is the input-channel threat (§6, threat 12): tool responses are an input channel through which the model's conclusion can be steered — including by content injected into retrieved material — and a single opaque blob digest gives a reviewer no practical way to inspect that channel. The base protocol retains the blob default for unregulated and low-stakes use; regulated profiles should not.

Replay classes. A `reason` step's `replay_class` declares the strength of replay claim the producer is making:

- R1 — Recorded only. The model identifier and conditions are recorded, but the producer is not claiming the step can be re-executed by a verifier. `output_artifact` MUST be present so that reviewers can inspect what was actually produced. No replay is performed; verification consists of structural and cryptographic checks plus inspection of the recorded artifact.
- R2 — Re-executable. The model is accessible to the verifier (by API, by hosted endpoint, or by other profile-recognized means) at a version matching `model.version`. Replay is permitted, but divergence between the replayed output and `output_hash` is expected to be possible and is not a verification failure (§3.3).
- R3 — Reproducible. The model weights are content-addressed by `model.weights_hash`, the runtime, tokenizer, decoding parameters, and (where applicable) random seed are pinned in `sampling` and `environment`, and the producer claims that re-execution by any party with access to the weights will yield bit-identical output. A `reason` step claiming R3 is held to a stronger verification standard (§3.2) than R1 or R2.

Aspirational status of R3. R3 is currently reachable for open-weight models executed on profile-recognized runtimes. Closed-weight hosted models — including most large language models accessed via commercial APIs as of this draft — cannot satisfy R3 because their weights are not content-addressable to the verifier. Producers building analyses on such models will operate at R2 by necessity. The protocol records the structural distinction so that, as the ecosystem of open-weight models accessible at the necessary scale develops, analyses can migrate from R2 to R3 without protocol changes. R3 is therefore present in the specification as a forward target as much as a present capability. Conformance levels (§5) account for this distinction directly: L4A (independently attested) admits R2 throughout; L4R (reproducible) requires R3 for designated high-stakes outputs.

R2 claims decay. An R2 claim is a claim about the *present* accessibility of a hosted model, and hosted models are versioned, deprecated, and withdrawn on commercial timescales that are short relative to regulatory review horizons. A proof produced today against a hosted model may be reviewed a year later, by which time the recorded `model.version` may no longer be resolvable; at that point the step degrades, without any fault in the proof, to linkage-verifiable basis (§3.2). Producers of proofs intended for long-horizon regulated review SHOULD therefore treat full artifact carriage — recorded output, input messages, tool-call log, and visible rationale all resolvable as bytes, ideally within an archival bundle (§2.8) — as the durable evidentiary posture, with R2 replay as a bonus available while the model remains accessible. Profiles SHOULD state the expected review horizon for their regulated scope and SHOULD require archival completeness (§2.8) where that horizon exceeds typical hosted-model availability. The verifier's report (§3.5) records, per step, whether replay was achieved or the step was evaluated on recorded artifacts alone.

The promotion of `reason` to a first-class step type — distinct from `compute`, with its own predecessor relations,

its own replay regime, its own visible-rationale evidence field, and its own verification semantics — is the principal way in which PoI extends prior provenance frameworks.

2.2.4 **attest** An `attest` step records a signed claim *about* one or more predecessor steps. It introduces no new derived output and produces no `output_hash`.

```
payload = {
  "claim_type": <claim-type identifier from a registered vocabulary; see below>,
  "role":       <attestor role identifier>,
  "claim_body": <inline structured claim object | reference>,
  "claim_hash": <digest of canonical encoding of claim_body>
}
```

An `attest` step MUST have at least one predecessor edge, and every predecessor edge MUST have relation `about`. An `attest` step MUST NOT have predecessor edges of any other relation.

Claim-type identifiers. A `claim_type` is either (a) an absolute URI, or (b) a compact name of the form `<family>/<name>` (e.g., `review/approve`), resolved against the base URI declared by the registering profile. The two forms are equivalent once resolved; verifiers MUST resolve compact names through the profiles named in the manifest. (v0.6.2 declared `claim_type` to be a URI while exemplifying compact names; this version legitimizes the compact form rather than pretending it away.) The base protocol mandates only that a resolved `claim_type` identify a definition specifying the structure of `claim_body` and the roles authorized to make the claim.

`attest` steps unify several concepts that are commonly designed as separate features. Claim types defined or anticipated for profile-level registration include:

- Review and approval. `review/approve`, `review/conditional`, `review/reject` by `qualified-reviewer` roles, used to record human review of `reason` or `compute` steps.
- Independent validation. `validation/replay-confirmed`, `validation/output-confirmed` by `independent-validator` roles.
- Qualification. `qualification/data-quality`, `qualification/vendor-status` by `data-provider` or `vendor-qualification` roles; `qualification/redaction-applied` by roles authorized to apply registered redaction policies (§2.2.5).
- Prespecification. `prespecification/locked-plan`, `prespecification/locked-protocol`, `prespecification/locked-charter` by `analysis-plan-author`, `biostatistician`, `model-owner`, or analogous roles. The `claim_body` identifies a pre-locked external artifact (statistical analysis plan, study protocol, model risk management policy, charter) and includes its content digest, the timestamp at which it was locked, lock evidence (an external trusted timestamp over the plan digest, where available), and the signatures of the authorizing parties at lock time. A prespecification attestation records that a downstream `compute` or `reason` step was performed against a plan that existed and was locked before the data underlying the step was unblinded or otherwise observable to the analyst. Prespecification is the protocol-level mechanism by which a reviewer can distinguish between ″this analysis was prespecified″ and ″this analysis was constructed to fit the observed result.″ The protocol provides the binding; the regulatory significance of that binding is a profile concern (§7.4). Where the locked plan enumerates an *analysis inventory*, the coverage mechanism of §5.6 additionally applies; the `claim_body` then carries an `analysis_id` binding the attested step to one inventory entry.
- Supersession. `supersession/retract` and `supersession/replace` as defined in §5.4.
- Adequacy and conditions. `adequacy/finding-confirmed`, `adequacy/finding-disputed`, recording independent review of a `reason` step's `finding_type` determination (§5.5).

This list is illustrative. The set of registered claim types is the protocol's primary extensibility surface; profiles add or refine claim types without modifying the base step taxonomy or verification algorithm.

Manifest-level attestations — claims about a proof as a whole rather than about any single step — are specified in §2.9, completing a deferral from v0.6.2.

2.2.5 Disclosure-limited artifacts (redaction) Regulated analyses routinely consume and produce content that cannot be disclosed to every verifier — protected health information in clinical input messages being the canonical case. v0.6.2 acknowledged this with an unspecified `redaction_policy` field; that underspecification was untenable, because the naive readings are both wrong: hashing *after* redaction breaks the bind between the recorded evidence and what the operation actually consumed or produced, while hashing *before* redaction makes the digest unverifiable by any verifier denied the unredacted bytes — and the spec said nothing about which claims survive in that state.

v0.7.0 resolves this with a dual-commitment carrier. Wherever a payload carries an artifact (`input_messages`, `output_artifact`, `tool_call_log`, `visible_rationale`, and `compute output_artifact`), the carrier takes one of three forms:

```
artifact_carrier =
    <inline canonical artifact>
  | { "uri": <URI>, "digest": <digest> }                    // content-addressed reference
  | {                                                        // disclosure-limited form
      "binding_digest":   <digest of canonical encoding of the UNREDACTED artifact>,
      "disclosed":        <inline canonical redacted artifact | content-addressed reference>,
      "disclosed_digest": <digest of canonical encoding of the redacted artifact>,
      "policy":           <identifier or content-addressed reference of the registered
                           redaction policy applied>
    }
```

Rules:

1. The payload's corresponding `*_hash` field (e.g., `input_messages_hash`) MUST equal `binding_digest`. The hash chain, invocation bindings, and all linkage checks bind to the unredacted content. Redaction never alters what the evidence *is*; it limits what is *disclosed*.
2. Where this specification requires that the verifier "be able to obtain" an artifact, a disclosure-limited carrier satisfies the requirement by yielding the disclosed form. The verifier MUST confirm the disclosed form digests to `disclosed_digest`.
3. A verifier without access to the unredacted bytes cannot recompute `binding_digest` and cannot perform replay or input-reconstruction checks that require the unredacted content. For such checks the step is *disclosure-limited*: it contributes linkage-verifiable basis at most, and the verification report (§3.5) MUST record the disclosure-limited status per step and per check.
4. An *authorized verifier* — one granted access to unredacted artifacts under the profile's access model — MAY obtain the unredacted bytes, MUST confirm they digest to `binding_digest`, and thereby upgrades the affected checks to their full strength. The protocol thus supports a two-tier review reality: most verifiers verify structure and linkage against redacted disclosures; authorized verifiers (e.g., a regulator operating under its own access authority) verify everything.
5. Redaction MUST be performed under a policy registered with a profile named in the manifest. Profiles SHOULD require a `qualification/redaction-applied` attestation, by a role authorized for the policy, about any step carrying disclosure-limited artifacts. The attestation is what makes the redaction itself reviewable: someone identifiable asserts that the disclosed form was produced from the bound form under the named policy. The protocol cannot mechanically verify that assertion without the unredacted bytes; the attestation exists so that the assertion is signed, not anonymous. (Threat 13, §6.)
6. The `redactions` record in a payload enumerates which of its artifact fields are disclosure-limited and under which policy; it exists for reviewer legibility and MUST be consistent with the carriers actually used.

What this mechanism deliberately does not include: per-field salted commitments, Merkle-structured selective disclosure, or zero-knowledge openings. Those designs allow an unauthorized verifier to confirm individual disclosed fields against the binding digest, at substantial encoding and tooling cost. They are deferred (Appendix E) until a profile demonstrates a concrete need that the dual-commitment-plus-authorized-verifier model does not meet; the protocol's regulated target users — regulators — characteristically possess

the access authority that makes the simpler model sufficient.

2.3 Edge relations

Every entry in a step's `predecessors` array is an edge. The protocol admits two equivalent surface forms for edges. The compact form is a two-field object:

```
edge = {
  "step":     <identity digest of predecessor step>,
  "relation": "derived-from" | "conditioned-on" | "about"
}
```

The extended form, used only with `conditioned-on` edges, optionally pins the role and a declared-relevance digest of the contextual relation:

```
edge = {
  "step":     <identity digest of predecessor step>,
  "relation": "conditioned-on",
  "context_role":            "policy" | "prior-finding" |
                             "review-context" | "analysis-plan" | "other"
                             | <profile-defined string>,
  "declared_relevance_hash": <digest of a short producer-supplied statement
                             of why the predecessor is relevant context>
}
```

The extended form is OPTIONAL in the base protocol. Profiles MAY require it for `conditioned-on` edges within their regulated scope. When the extended form is used, the verifier MUST canonically encode the full edge object (including the extended fields) when computing the step identity. The extended fields are not subject to replay verification — they discipline what the producer asserts about the contextual relation, they do not establish operational influence.

The permitted relations per step type are summarized below. A proof in which a step has a predecessor edge with a relation not permitted for its type is malformed and MUST fail verification.

Step type	`derived-from`	`conditioned-on`	`about`	Min predecessors
`observe`	not permitted	not permitted	not permitted	0
`compute`	required	not permitted	not permitted	≥1
`reason`	permitted	permitted	not permitted	≥1
`attest`	not permitted	not permitted	required	≥1

A step MUST NOT reference the same predecessor through more than one edge. (For a `reason` step, `derived-from` subsumes the contextual assertion of `conditioned-on`; carrying both is redundant and is prohibited to keep the test suite's equality checks over predecessor sets unambiguous.)

The semantics of each relation are:

- **`derived-from`**: the predecessor's output is part of the input. For a `compute` step, this means the predecessor's output is consumed by the function. For a `reason` step, this means the predecessor's output is part of the prompt or system input. A `derived-from` predecessor MUST be a step type for which `STEP_RESOLVE` (§3.0) is defined; it MUST NOT be an `attest` step.
- **`conditioned-on`**: the predecessor is declared as evidentiary context relevant to the current step but is not a direct input to the operation. Common cases include a prior `reason` step whose conclusion frames the current step's question through producer-side workflow logic, or a policy or charter asserted as governing the analysis but not literally bound into the model input. The protocol records `conditioned-on` predecessors as part of the invocation but does not establish how, or whether, they affected the operation; their inclusion is an attestor assertion that may be surfaced for human review or compliance

predicates. Where the extended form is used, the assertion is further pinned by `context_role` and `declared_relevance_hash`, giving reviewers a producer-supplied basis for the contextual relation without making the relation replay-verifiable. Distinguishing `conditioned-on` from `derived-from` matters because the verifier's replay procedure differs (§3.2).

- **`about`**: the current step asserts something *regarding* the predecessor. The current step does not depend on the predecessor's output for its own derivation; it claims something *about* it.

2.4 Timestamp binding

Every step MUST carry a `timestamp` field of the following form:

```
timestamp = {
  "value":     <RFC 3339 timestamp>,
  "authority": <URI identifying the timestamp authority>,
  "token":     <opaque token from the timestamp authority>
}
```

The timestamp authority attests to the step's identity — `digest(to_timestamp(step))`, per §2.5 — which by preimage commits to the full signed step content including the attestor's signature. The timestamp authority's binding mechanism is profile-defined. A profile MUST specify how a verifier confirms that `token` represents a valid attestation by `authority` that the step identity existed at `value`, and MUST ensure that both `authority` and `value` are bound within the material the token verification covers (RFC 3161 tokens bind the generation time internally; transparency-log proofs bind the integrated time; profiles binding other authority types must achieve the equivalent). A token that verifies only the digest, leaving `value` unauthenticated, does not satisfy this requirement.

PoI does not mandate any single timestamp authority. Acceptable authority types include but are not limited to: RFC 3161 timestamping services, Sigstore Rekor inclusion proofs, public-blockchain anchors, and notarial attestations. Profiles bind specific authority types and specify their verification procedures (§7).

Ordering and skew. A timestamp on a step MUST NOT precede the timestamps of any of its predecessors by more than the skew tolerance δ. A verifier MUST check, as part of structural validation (§3.1), that for every edge from step s to predecessor s':

$$\texttt{timestamp(s').value} \leq \texttt{timestamp(s).value} + \delta$$

Profiles MUST specify δ for their regulated scope; in the absence of a profile value, verifiers SHALL use δ = 300 seconds. The tolerance exists because steps in one proof may be timestamped by different authorities whose clocks legitimately disagree by seconds; a hard zero-tolerance check (as in v0.6.2) fails honest proofs on clock skew while defending nothing additional — the *logical* ordering of steps is already proven, independently of any clock, by the hash chain itself (a step's identity commits to its predecessors' identities, so a predecessor's signed content necessarily existed before its successor was encoded). The wall-clock check is a heuristic defense against gross backdating and a sanity check on authority behavior; it is not the source of the protocol's ordering guarantee, and verifiers' diagnostics SHOULD say so when the check fails.

2.5 Canonical encoding and identity

A step's canonical encoding is the JSON representation of the step record using RFC 8785 (JSON Canonicalization Scheme). All digests referenced in this document are computed over the canonical encoding of the relevant object, per the rules of this section and §2.5.1.

A step's identity is computed as:

```
id(step) = digest( to_timestamp(step) )       // fields 1..6, including signature,
                                              // excluding timestamp
```

The identity is determined after the attestor has signed the step and before any timestamp is obtained; the timestamp authority then attests to the identity (§2.4). The identity is carried as a digest object (§1.5); references to predecessor identities in edges and invocations carry the digest object in full, so that the digest

algorithm of every link in the chain is explicit. The design trade-offs of excluding the timestamp from the identity are stated in §2.1.

2.5.1 Canonical artifact encoding Digests over artifacts — observed content, invocation objects, input messages, tool-call logs, rationales, claim bodies, and operation outputs — are computed over the artifact's *canonical artifact encoding*, defined by encoding identifier:

- **`jcs+json`** — the artifact is a JSON value; its canonical artifact encoding is its RFC 8785 encoding. JSON cannot represent non-finite numbers (NaN, ±Infinity); an operation whose natural output contains such values MUST NOT use `jcs+json` for that output.
- **`octet-stream`** — the artifact is an arbitrary byte sequence; its canonical artifact encoding is the byte sequence itself, unmodified. The artifact's type is conveyed by the surrounding context (`content_type` for observed artifacts; the function or model contract for outputs).
- Profiles MAY register additional encoding identifiers (e.g., for columnar dataset formats with defined canonical serializations).

For `observe` steps, `content_hash` is computed under `octet-stream` over the observed bytes unless the profile specifies otherwise. For `compute` and `reason` steps, the REQUIRED `output_encoding` field declares which rule governs `output_hash`; the replayed output is encoded under the same rule before comparison. Invocation objects, reason invocations, input message sequences, edges, manifests, and claim bodies are JSON values and are always encoded under `jcs+json`.

This subsection closes a normative gap in v0.6.2, which referred to a "canonical output encoding" without defining one for non-JSON outputs. The gap was not cosmetic: bit-identical replay is undefined unless producer and verifier deterministically agree on output bytes, and statistical outputs (floating-point payloads, datasets, images) are precisely where the disagreement bites.

2.5.2 Digest comparison For digests, comparison is exact: digest objects match if and only if their `alg` and `value` fields are equal. There is no notion of digest equality "within tolerance." Where tolerance replay is required (numerical equivalence under non-deterministic floating-point reductions, for example), the protocol carries the recorded output as an artifact and applies a profile-defined equivalence predicate to the *outputs*, not to the digests. This separation — exact comparison for digests, predicate comparison for outputs — is the protocol's way of admitting numerical practicality without compromising the integrity guarantees of content-addressing.

2.6 Construction-time invariants

A step is well-formed if and only if:

1. Its `type` is one of the four defined values.
2. Its `predecessors` array satisfies the relation constraints of §2.3 for its type, including the prohibition on `attest` steps appearing as `derived-from` predecessors and the prohibition on multiple edges to the same predecessor. Edges of relation `conditioned-on` MAY use the extended form (§2.3); edges of other relations MUST use the compact form.
3. Its `payload` matches the schema for its type (§2.2), including the presence of `invocation` (and, for `reason`, `input_messages`) as an admissible artifact carrier (§2.2.5), the presence of `output_encoding` on `compute` and `reason`, and the internal consistency of any disclosure-limited carriers with the `redactions` record.
4. Its `timestamp` is structurally valid and satisfies the ordering-with-skew rule of §2.4 with respect to every predecessor.
5. Its `signature` verifies against the public key bound to `attestor`.
6. Each predecessor referenced exists in the proof, and the proof is acyclic when traversed via `predecessors` edges.

A proof is well-formed if and only if every step in it is well-formed and condition (6) holds globally. Well-formedness is necessary but not sufficient for verification; verification additionally applies type-specific checks and conformance-level checks (§3, §5).

2.7 The proof manifest

A PoI proof is not transmitted or verified as a bare collection of steps. It is carried alongside a *proof manifest*, a top-level object that names the steps comprising the proof, identifies the output steps whose outputs are claimed as conclusions, declares the conformance level being asserted, and binds the proof to one or more profiles under which it is to be verified.

```
manifest = {
  "manifest_version":     "0.7.0",
  "proof_id":             <UUID or content-derived identifier>,
  "steps":                [ <step identity digest>, ... ],
  "outputs":              [ <step identity digest>, ... ],
  "conformance_claim":    "L1" | "L2" | "L3" | "L4A" | "L4R",
  "verification_basis":   <optional> "replay-verifiable" |
                                     "linkage-verifiable-only" |
                                     "resolution-limited",
  "profiles":             [ <profile URI>, ... ],
  "manifest_attestor":    <URI identifying the producer claiming this proof>,
  "manifest_signature":   <signature object over canonical encoding of preceding fields>
}
```

The manifest's `outputs` field replaces the parameter `O` in §3.1's structural-validation algorithm. Each identity in `outputs` MUST identify a step of type `compute` or `reason` unless a profile explicitly permits otherwise. `observe` steps carry external data rather than derived output; `attest` steps carry claims rather than output (§3.0). Permitting them as outputs without profile authorization would create ambiguity in conformance evaluation.

The manifest is signed by the producer asserting the conformance claim. The manifest signature is structurally identical to a step signature but is *not* itself an Insight Step — it carries no step type, no predecessor edges, and does not appear in the proof DAG. This separation resolves an issue that would otherwise arise: a conformance claim made via an `attest` step inside the proof would require the proof to contain `attest` steps even when claiming a level (L1) that prohibits them. The manifest sits outside the typed-step DAG specifically to avoid this circularity.

The manifest's *identity* is the digest of its canonical encoding including `manifest_signature`; manifest-level attestations (§2.9) and verification reports (§3.5) reference the manifest by this digest.

Verification basis. The OPTIONAL `verification_basis` field records the producer's claim about what gate of verification the proof supports:

- **`replay-verifiable`**: the producer asserts that every `compute` step is replayable under its declared regime and every `reason` step is verifiable under its declared replay class, given the profile-defined resolution services. A verifier with full resolution access SHOULD be able to perform gates 1–5 (Appendix A).
- **`linkage-verifiable-only`**: the producer asserts that the proof supports gates 1–3 only (schema, structural, cryptographic), and does not undertake to make functions, models, or weights resolvable to the verifier. Replay is not claimed.
- **`resolution-limited`**: the producer asserts a mixed state — some steps support replay, others do not — and acknowledges that the achievable verification basis depends on the verifier's resolution capabilities and on which artifacts are supplied alongside the proof.

The field is informational at the base protocol level. Profiles MAY require a specific `verification_basis` value as a precondition for use of the proof in their regulated scope. A verifier MUST report, alongside its accept/reject decision, the verification basis it actually achieved (which may be weaker than the claimed basis if resolution services are unavailable). When the achieved basis is weaker than the claimed basis, the verifier MUST identify the steps for which the gap arose. The report format is specified in §3.5.

The absence of `verification_basis` is permitted; in that case the verifier treats the basis as unspecified and reports its achieved basis without comparison.

2.8 Archival bundles and offline verification

A proof whose verification depends on producer-operated resolution services is, in practice, a proof whose verifiability the producer controls and whose verifiability decays as those services change. Content-addressing already reduces a malicious resolution service to an *availability* attack — it can withhold artifacts but cannot substitute them undetectably — and hosted-model deprecation makes even honest availability time-limited (§2.2.3, "R2 claims decay"). Regulated review additionally occurs, as a practical matter, in environments where reviewers do not and should not call producer endpoints during evaluation.

The *archival bundle* is the protocol's answer: a self-contained, content-addressed packaging of a proof such that verification through gate 4 — excluding live model replay — requires no network access.

A bundle is a directory tree (or an archive of one) with the following layout:

```
bundle/
  bundle.json                      // bundle manifest (below)
  manifest.json                    // the proof manifest (§2.7)
  steps/<id-alg>/<id-value>.json   // one file per step, named by identity digest
  artifacts/<alg>/<value>          // content-addressed artifact store: every
                                   // referenced artifact, stored by digest of its
                                   // canonical artifact encoding
  attestations/<id>.json           // manifest-level attestations (§2.9), if any
  profiles/<alg>/<value>           // snapshots of the profile documents named in
                                   // the manifest, stored by digest
```

```
bundle_manifest = {
  "bundle_version":  "0.7.0",
  "manifest_digest": <digest of the proof manifest>,
  "contents":        [ { "path": <relative path>, "digest": <digest> }, ... ],
  "completeness":    "archival-complete" | "partial",
  "gaps":            <required when partial: list of { "step": <id>,
                      "field": <payload field>, "digest": <digest>,
                      "reason": <string> } enumerating every referenced
                      artifact not present in the store>,
  "bundle_attestor":  <URI>,
  "bundle_signature": <signature object over canonical encoding of preceding fields>
}
```

A bundle is **archival-complete** if and only if every digest referenced by any step in the structural ancestor closure of the manifest's outputs — observed artifacts, invocation objects, input messages, tool-call logs, visible rationales, output artifacts, claim bodies, and, for disclosure-limited carriers, the *disclosed* forms — resolves within the bundle's artifact store. (Unredacted forms behind disclosure-limited carriers are excluded from the completeness requirement by construction; their availability is governed by the profile's access model, and profiles MAY define a separate authorized-bundle layout that includes them for authorized verifiers.) Model weights are excluded from the completeness requirement; where R3 steps are present, the bundle records the weights digest and the profile's weights-resolution mechanism applies.

Rules:

1. Within a bundle, content-addressed reference resolution (§2.2.2, §2.2.3) is against the bundle's artifact store. A verifier verifying a bundle MUST NOT require network access for gates 1–4 except for live model replay (R2/R3) and except for trust material (keys, revocation records, authority roots) where the profile has not provided an offline trust snapshot; profiles intended for regulated submission SHOULD specify an offline trust snapshot format.
2. A verifier MUST independently confirm the bundle's completeness declaration: it recomputes the referenced-digest set from the steps and checks the store, rather than trusting `completeness`. The achieved completeness is recorded in the verification report (§3.5). A false `archival-complete` declaration is a producer misrepresentation analogous to verification-basis overclaiming (§6, threat 10).

3. The bundle is the RECOMMENDED interchange format for proofs submitted to regulated review. Where a profile's regulated scope involves review horizons exceeding typical hosted-service availability, the profile SHOULD require archival-complete bundles.

The bundle specifies *packaging*, not *retention* (§1.3): it makes long-horizon verification possible; it does not say who must keep the bundle or for how long.

2.9 Manifest-level attestations

A producer or external party may need to attest about a proof as a whole — for example, a quality-assurance sign-off over an entire submission package — rather than about any single step. v0.6.2 deferred this; v0.7.0 specifies it.

A *manifest-level attestation* is a separately-signed sibling object to the manifest. It is not an Insight Step, carries no edges, and does not appear in the proof DAG:

```
manifest_attestation = {
  "version":    "0.7.0",
  "subject":    { "proof_id": <as in manifest>, "manifest_digest": <digest> },
  "claim_type": <claim-type identifier, as in §2.2.4>,
  "role":       <attestor role identifier>,
  "claim_body": <inline structured claim object | reference>,
  "claim_hash": <digest of canonical encoding of claim_body>,
  "attestor":   <URI>,
  "signature":  <signature object over canonical encoding of preceding fields>,
  "timestamp":  <timestamp object (§2.4) binding to the attestation's identity>
}
```

A manifest-level attestation's identity is the digest of its canonical encoding excluding `timestamp`, mirroring the step rule (§2.5). Verification mirrors the `attest` checks of §3.2: signature, timestamp, claim-digest consistency, and role authorization under the profile's authority model as of the attestation's timestamp. The `subject.manifest_digest` binds the claim to one specific signed manifest; an attestation whose subject digest does not match the manifest under verification MUST be disregarded with a diagnostic.

Manifest-level attestations are carried in the bundle's `attestations/` directory (§2.8). They do not participate in conformance-level predicates of §5 unless a profile says otherwise; their canonical use is profile-defined admissibility conditions over whole proofs.

---

3. Verification

This section specifies the verification algorithm. The algorithm is a single procedure parameterized by the conformance level being checked. For a fixed proof, fixed manifest, fixed profile set, fixed external artifact set, fixed trust roots, and fixed replay configuration, compliant verifier implementations MUST produce the same accept/reject decision; verifiers MAY differ in the diagnostics they emit on rejection. Where replay configurations differ (for example, where one verifier has access to a model that another does not), the resulting decisions MAY differ in well-defined ways; the protocol distinguishes between failures that arise from the proof itself and failures that arise from verifier-side resolution limits, and each `FAIL` diagnostic identifies its source.

A verifier MUST report, alongside its accept/reject decision, the verification basis (§2.7) it actually achieved, in the verification report format of §3.5. Where the manifest carries a `verification_basis` claim and the achieved basis is weaker, the verifier MUST identify the gap (which steps did not reach replay-verifiable status, and why).

Authorization is evaluated as of signing time. Every authority check in this section — observe-source authorization, attest role authorization, redaction-policy authorization — evaluates the attestor's authorization

status *as of the step's `timestamp.value`*, not as of verification time. A profile MUST provide a mechanism for resolving historical authorization (e.g., dated role registries, revocation records with effective dates, transparency-logged credential issuance). The rationale: regulated proofs are reviewed months or years after production; evaluating authority at verification time would make honest old proofs unverifiable whenever personnel change or keys rotate, and — worse — would validate claims signed *after* a revocation if the credential were later reinstated. Key compromise discovered after the fact (where signatures made during the compromise window carry timestamps inside it) is threat 2 in §6; the protocol surfaces the window, profiles define the remediation.

3.0 The **`STEP_RESOLVE`** operation

The verification algorithm uses an internal operation `STEP_RESOLVE(s)`, returning the "output" of a step `s`. Its behavior is defined per step type:

- **`observe`**: returns the observed artifact, identified by `payload.content_hash`. If the artifact itself is not supplied to the verifier (directly, or within a bundle's artifact store), `STEP_RESOLVE` returns `content_hash` as an opaque reference; downstream linkage checks confirm digest equality without requiring artifact retrieval. The verifier MUST track which form was returned for downstream diagnostics.
- **`compute`**: returns the output, identified by `payload.output_hash`. The output bytes are retrievable from `payload.output_artifact` if present; otherwise from replay if replay is enabled and succeeds; otherwise `STEP_RESOLVE` returns `output_hash` as an opaque reference.
- **`reason`**: returns the output, identified by `payload.output_hash`. The output bytes are retrievable from `payload.output_artifact` if present (REQUIRED at R1, RECOMMENDED at R2 and R3 when the proof is intended to be reviewable without replay). Otherwise `STEP_RESOLVE` returns `output_hash` as an opaque reference.
- **`attest`**: `STEP_RESOLVE` is undefined. `attest` steps record claims, not derived outputs. An `attest` step MUST NOT appear as a `derived-from` predecessor of any step (§2.3, §2.6). A proof violating this constraint MUST fail well-formedness.

A step "resolves to bytes" if `STEP_RESOLVE` returns concrete content rather than an opaque digest reference. Where the content is carried in a disclosure-limited form (§2.2.5), `STEP_RESOLVE` returns the *disclosed* bytes flagged `disclosure-limited`; an authorized verifier with access to the unredacted form returns the unredacted bytes flagged `full`. Replay verification (§3.2) requires that all `derived-from` predecessors of the step under examination resolve to bytes flagged `full`; where any resolves only to disclosed or opaque form, the step is verifiable in the weaker sense of structural, signature-, and linkage-validity only, without replay. The verifier records which sense applied per step for the verification report (§3.5).

3.1 Structural validation

Given a proof manifest `M` (§2.7) and the set of steps `P` it references, the verifier first validates the manifest itself and then the proof DAG:

```
STRUCTURAL_VALIDATE(M, P):
    0. Verify M.manifest_signature against the public key bound to
       M.manifest_attestor. On failure: FAIL("manifest signature invalid").
       Confirm that the set of step identities in M.steps equals the
       set of identities computed for P. On mismatch:
         FAIL("manifest does not describe proof").
       Confirm that every identity in M.outputs is present in M.steps.
       On mismatch: FAIL("output not in proof").
       Confirm that every output step is of type `compute` or `reason`,
       except where the named profiles explicitly permit other types as
       outputs. On violation: FAIL("output of impermissible type").
    1. For each step s ∈ P:
         if not WELL_FORMED(s):  return FAIL("step ill-formed", s)
    2. Build the predecessor graph G over P. If G contains a cycle:
```

```
      return FAIL("proof contains cycle")
  3. For each edge (s → s') in G:
      if s' ∉ P:                return FAIL("dangling predecessor", s, s')
      if timestamp(s').value > timestamp(s).value + δ:
                                return FAIL("timestamp inversion beyond
                                            skew tolerance", s, s')
      if relation = "derived-from" and type(s') = "attest":
                                return FAIL("attest cannot be derived-from",
                                            s, s')
  4. Let O = M.outputs.
  5. Compute the structural ancestor closure A = transitive closure of
     predecessors of O. This closure includes all steps regardless of
     supersession status; the protocol does not erase the historical
     derivation of any output. Steps in P \ A are "unreached." A
     verifier MAY emit a warning for these but MUST NOT reject on
     this basis.
  6. Compute the effective ancestor closure A* = A minus any step
     superseded by an attest of claim_type "supersession/retract" or
     "supersession/replace" (§5.4). A* is the closure over which
     conformance checks of §5 are evaluated.
  7. For each output step o ∈ O: if any step in the structural
     ancestor closure of o is superseded but o itself is not
     superseded, return FAIL("output derived from superseded ancestor
     not itself superseded", o). A producer correcting an analysis
     MUST supersede both the retracted predecessor and any downstream
     output that depends on it (or replace the output with a non-
     superseded successor); see §5.4.
  8. Return PASS.
```

`WELL_FORMED(s)` is the conjunction of conditions 1–5 of §2.6 (condition 6 is checked globally in steps 2–3 above). δ is the skew tolerance of §2.4.

3.2 Type-specific verification

Given a structurally valid proof, the verifier walks the steps in topological order (predecessors before successors) and applies type-specific checks. All authority checks evaluate authorization as of the step's timestamp (§3 preamble).

```
TYPE_VERIFY(s):
    case s.type of:

      observe:
        a. If an external artifact for s is supplied (directly or via
           bundle store), recompute the digest of its canonical artifact
           encoding and compare with payload.content_hash. On mismatch: FAIL.
        b. Resolve s.attestor under the profile's authority model, as of
           s.timestamp.value. If the attestor was not authorized to make
           observations from payload.source at that time: FAIL.
        c. If payload.provenance references an external attestation,
           verify it using the relevant external verifier. On failure: FAIL.

      compute:
        a. Obtain the canonical invocation object: either from
           s.payload.invocation inline, or by resolving s.payload.invocation
           as a content-addressed reference (profile-defined resolution;
```

```
      bundle store within a bundle). Compute the digest of its
      canonical encoding. If the result does not equal
      s.payload.invocation_hash: FAIL.
  b. Confirm that the invocation object's `inputs` list, when matched
      by step identity, equals the set of `derived-from` predecessors
      of s. For each entry (name, step, output_hash) in inputs,
      confirm that output_hash matches the predecessor's recorded
      output_hash. On any mismatch: FAIL.
  c. Resolve s.payload.function. If the function is not resolvable,
      replay is impossible and the step is verifiable in a weaker
      sense only: it remains structurally, signature-, and linkage-
      verified, but no replay claim is made. The verifier records
      this with diagnostic `compute: function-unresolvable` and
      accounts for the step under linkage-verifiable-only basis.
  d. If replay is enabled, the function is resolvable, and every
      `derived-from` predecessor resolves to bytes flagged `full`:
        - If environment.replay_regime = "bit-identical":
            re-execute the function on the resolved inputs in an
            environment matching s.payload.environment. Encode the
            replayed output under s.payload.output_encoding and
            compute its digest. If it does not equal
            s.payload.output_hash: FAIL.
        - If environment.replay_regime = "tolerance":
            s.payload.output_artifact MUST be present. Re-execute
            the function on the resolved inputs. Apply the
            equivalence predicate named in s.payload.environment to
            the pair (replayed output, recorded output_artifact).
            If the predicate returns false: FAIL. The verifier MAY
            additionally digest the recorded output_artifact and
            confirm it matches output_hash; this confirms the
            artifact has not been tampered with relative to the
            signed step but does not by itself verify replay.
      If any derived-from predecessor resolves only to disclosed
      (disclosure-limited) or opaque form, record
      `compute: replay-blocked, inputs-not-fully-resolvable` and
      account the step under linkage-verifiable-only basis. An
      authorized verifier with full access repeats the check at
      full strength.

reason:
  a. Obtain the canonical reason-invocation object from
      s.payload.invocation (inline or by resolving the reference).
      Compute the digest of its canonical encoding. If it does not
      equal s.payload.invocation_hash: FAIL.
  b. Confirm that the invocation object's input_bindings list, when
      matched by step identity, equals the set of `derived-from`
      predecessors of s, and that its context_frame.conditioned_on
      list equals the set of `conditioned-on` predecessors of s. For
      each input binding (name, step, output_hash), confirm that
      output_hash matches the predecessor's recorded output_hash.
      On any mismatch: FAIL.
  c. Obtain the input message sequence from s.payload.input_messages.
        - Inline or content-addressed: compute the digest of its
          canonical encoding; if it does not equal
```

s.payload.input_messages_hash: FAIL.
- Disclosure-limited (§2.2.5): obtain the disclosed form; compute its digest; if it does not equal disclosed_digest: FAIL. Record the field as disclosure-limited. An authorized verifier additionally obtains the unredacted form and confirms its digest equals binding_digest (= input_messages_hash); on mismatch: FAIL.

d. (Optional, recommended; requires bytes flagged `full`.) Reconstruct the input messages from the resolved `derived-from` predecessor outputs using the binding names recorded in the invocation, and confirm the reconstruction is consistent with the resolved input_messages. The base protocol does not normatively specify a single reconstruction rule because the templating and message-assembly conventions are model- and producer-specific; profiles MAY require a specific reconstruction predicate.
e. If tool_call_log_hash is present and a tool_call_log carrier is supplied, confirm the carrier digests to tool_call_log_hash (disclosed_digest for disclosure-limited carriage). Likewise for visible_rationale_hash / visible_rationale. On mismatch: FAIL.
f. Dispatch on s.payload.replay_class:
    - R1 (recorded only):
      s.payload.output_artifact MUST be present. Digest the artifact and confirm it matches s.payload.output_hash (or disclosed_digest under disclosure-limited carriage, with binding confirmation reserved to authorized verifiers). No re-execution is performed. Record the step as `reason-class: R1, replay: not-attempted` and contributing linkage-verifiable basis only.
    - R2 (re-executable):
      If replay is enabled, s.payload.model is resolvable at the recorded version, and inputs resolve to bytes flagged `full`: re-execute under s.payload.sampling. Encode the replayed output under output_encoding, digest, and compare to s.payload.output_hash:
        - Match: record `reason-class: R2, replay: stable`.
        - Mismatch: record `reason-class: R2, replay: divergent` together with the replayed output digest. This is NOT a verification failure (§3.3).

      If the model is unresolvable, record `reason-class: R2, replay: model-unavailable` and contribute linkage-verifiable basis only. (See §2.2.3, "R2 claims decay": this diagnostic is the expected long-horizon state of honest hosted-model proofs, which is why full artifact carriage is the durable evidentiary posture.)
    - R3 (reproducible):
      s.payload.model.weights_hash MUST be present. The verifier MUST resolve weights by digest; if unable, FAIL with `reason-class: R3, replay: weights-unavailable`. Re-execute under the fully pinned conditions. Encode under output_encoding, digest, and compare to s.payload.output_hash. If it does not match: FAIL.

```
            (At R3, divergence IS a verification failure, because
            R3 asserts reproducibility.)

attest:
  a. Resolve all `about` predecessors. Confirm each is present in
     the proof. (Ill-formed proofs would have failed at step 1
     of STRUCTURAL_VALIDATE; this is a defensive check.)
  b. Resolve s.attestor and confirm the attestor's role
     (s.payload.role) was authorized — under the profile's
     authority model, as of s.timestamp.value — to make claims of
     type s.payload.claim_type about steps of the predecessor's
     type. On unauthorized: FAIL.
  c. Confirm s.payload.claim_hash matches the digest of the
     canonical encoding of s.payload.claim_body. On mismatch: FAIL.
```

3.3 The semantics of **reason** divergence

A `reason` step that re-executes to a different output than its recorded `output_hash` is divergent. Divergence is *information*, not a verdict. The protocol commits to recording divergence; what to do with the record is delegated to the conformance regime and to human reviewers.

This delegation is deliberate. Possible meanings of divergence include:

- **Stochasticity within tolerance.** The model is non-deterministic by design and the divergent output expresses substantively the same conclusion. Whether two outputs express the same conclusion is generally not mechanically decidable.
- **Material change in conclusion.** The divergent output expresses a conclusion that, if it had been recorded originally, would have led to a different downstream analysis. This is a finding for human review and may trigger a supersession (§2.2.4).
- **Model drift or unavailability.** The model identifier resolves to a model whose behavior has changed since the original step was produced. This is a finding for the producer's model-management process.

A single replay yields a single sample of a stochastic process; "stable" on one replay is weak evidence of determinism (hosted inference is non-deterministic under batching even at temperature zero), and "divergent" on one replay quantifies nothing. The base protocol deliberately records only the single-replay outcome and its digest; profiles MAY define multi-replay stability evidence (e.g., a producer-supplied attestation recording the distribution of k production-time replays) as a registered claim type, without base-protocol machinery. A richer base semantics for divergence — for example, typing outputs under composition rules that determine which downstream conclusions remain valid under which divergence conditions — remains out of scope for v0.7.0.

3.4 Complexity

For a proof with `n` steps, the graph-theoretic portion of verification — well-formedness checks, cycle detection by Kahn's algorithm, ancestor closure computation, supersession analysis, and the coverage check of §5.6 (which is a set-mapping check over claim bodies) — runs in `O(n + m)` time and space, where `m` is the total number of edges and inventory entries. Total verification cost is dominated by the additive cost of cryptographic operations (signature and timestamp verification, scaling per-step but constant in `n`), external resolution (function and model retrieval, content-addressed reference resolution, external provenance verification), artifact retrieval, replay execution itself (which for `reason` steps may be the dominant term), and profile predicate evaluation. The structural claim is that the graph-traversal cost is linear and tractable; the operational cost is dominated by terms outside the graph itself, which scale with the producer's resource decisions rather than with proof size.

3.5 The verification report

The requirement that a verifier "report" its decision, achieved basis, and gaps appeared in v0.6.2 with no defined carrier, which made the requirement untestable and left every verifier to invent an incompatible format. For a regulator, the verification report is not an implementation detail: it is itself the evidence artifact that attaches to a review record, and a standard format is what permits an ecosystem of independent verification services whose outputs are mutually legible.

A compliant verifier MUST be capable of emitting a verification report of the following form. The report is a JSON value, canonically encoded under `jcs+json`:

```
verification_report = {
  "report_version":    "0.7.0",
  "proof_id":          <as in manifest>,
  "manifest_digest":   <digest of the verified manifest>,
  "profiles_applied":  [ <profile URI>, ... ],
  "claimed_level":     <manifest.conformance_claim>,
  "result":            "PASS" | "FAIL",
  "failures":          [ <FAIL diagnostics with step identities and the
                          source classification: proof-defect |
                          resolution-limit>, ... ],
  "claimed_basis":     <manifest.verification_basis | "unspecified">,
  "achieved_basis":    "replay-verifiable" | "linkage-verifiable-only"
                       | "resolution-limited",
  "bundle":            <optional> { "bundle_digest": <digest>,
                        "declared_completeness": <as declared>,
                        "confirmed_completeness": "archival-complete" |
                        "partial", "gaps_confirmed": [...] },
  "coverage":          <optional, per §5.6> { "plans": [ { "plan_digest":
                        <digest>, "status": "satisfied" | "violated" |
                        "not-evaluable", "missing": [ <analysis_id>, ... ] } ] },
  "steps": [
    {
      "step":        <identity digest>,
      "type":        <step type>,
      "status":      "verified" | "failed",
      "basis":       "replay" | "linkage-only",
      "disclosure":  "full" | "disclosure-limited" | "opaque",
      "replay":      <optional, reason steps: "stable" | "divergent" |
                      "not-attempted" | "model-unavailable" |
                      "weights-unavailable", with replayed digest where
                      divergent>,
      "diagnostics": [ <strings> ]
    },
    ...
  ],
  "replay_configuration": <summary of resolution services, model access,
                           and authorization tier used>,
  "verifier":          <URI identifying the verifying party and implementation>,
  "generated_at":      <RFC 3339>,
  "verifier_signature": <optional signature object; SHOULD be present when
                         the report is itself used as evidence>
}
```

The report's required content is normative; its field set MAY be extended by profiles. A report used as evidence in a regulated process SHOULD be signed by the verifier and MAY itself be timestamped under

§2.4 semantics.

---

4. Properties (Derived)

This section derives the properties commonly demanded of regulated evidence — traceability, reproducibility, reviewability, robustness, regulatability, and (new in this version) accountability of prespecified scope — as consequences of the protocol specified in §2 and §3. Each property is stated as a proposition followed by a brief justification. None of these properties is a separate protocol feature; each follows from the construction of the Insight Step and the verification algorithm.

4.1 Traceability

For every step `s` not of type `observe`, every input to `s` is reachable, by transitive traversal of `predecessors` edges, to one or more `observe` steps.

This holds by induction on the predecessor graph: every non-`observe` step has at least one predecessor (§2.3), and the graph is acyclic (§2.6), so every backward path terminates at an `observe`. Traceability is therefore not an external property; it is a structural invariant of any well-formed proof.

4.2 Reproducibility

Every `compute` step is replayable under its declared replay regime — bit-identical, or output-equivalent under a profile-defined predicate. Every `reason` step is verifiable under its declared replay class — R1 (recorded only), R2 (re-executable with possible divergence), or R3 (reproducible bit-identically against pinned weights).

This follows directly from §2.2.2 and §2.2.3, with the qualification — made explicit in this version — that replay claims are claims about resolvability, and resolvability decays (§2.2.3). The protocol distinguishes deterministic and non-deterministic operations by step type, and within each, distinguishes the strength of the reproducibility claim being made. Reproducibility is not a single property but a regime-and-class lattice; the conformance levels of §5 select which regions of the lattice are admissible for which classes of analysis. The archival bundle (§2.8) is what makes the *recorded* leg of every claim durable independently of resolvability.

4.3 Reviewability

Every step's payload is independently inspectable given its predecessors and either the profile-defined resolution mechanism for its identifiers or an archival-complete bundle.

Content-addressing (§2.5) ensures that any inspector with the proof and access to the resolution services — or, equivalently and without network dependence, an archival-complete bundle (§2.8) — can recover the data described by every payload field, in disclosed form where disclosure-limited carriage applies (§2.2.5). No proprietary tooling is required. Visible rationale and tool-call logs (§2.2.3) are independently inspectable in the same way.

4.4 Robustness (tamper-evidence)

Any modification to any field of any step other than `timestamp` changes that step's identity, and any modification to a step's identity invalidates every descendant that references it. Any modification to a step's `timestamp` is detected by timestamp verification.

The first sentence follows from the definition of step identity as a digest over the canonical encoding of fields 1–6 (§2.5) and the use of identities (not positions or pointers) in `predecessors` edges (§2.3). The second follows from §2.4: the timestamp token is a self-authenticating attestation by the authority over the step identity, the authority URI, and the value; altering any of the three breaks token verification. The residual freedom — substituting one *genuine* token for another over the same identity — is analyzed in §2.1 and bounded there:

substitution can move evidence later in time but not earlier without authority compromise (threat 7). The cost of forging a content modification is the cost of producing a digest collision and of forging or coercing every signature on every descendant step.

4.5 Regulatability

Compliance rules expressible as predicates over typed steps, signed roles, claim bodies, and the manifest in the proof DAG can be mechanically checked against any well-formed proof.

The combination of typed steps (§2.2), typed edge relations (§2.3), role-bearing `attest` steps (§2.2.4), and structured claim bodies is sufficient to express predicates such as: ″every `reason` step on the path to a regulated conclusion has at least one `attest` of role `qualified-reviewer` about it,″ or ″every `compute` step performing a confirmatory analysis is bound by a `prespecification/locked-plan` attestation whose lock evidence predates the relevant unblinding event.″ A profile MAY supply a library of such predicates corresponding to a specific compliance regime (e.g., 21 CFR Part 11, SR 11-7, GMLP).

This is the precise form of the protocol's claim to support compliance: not that PoI implements any specific regulation, but that PoI's structure permits expressing compliance rules as decidable predicates over the proof. The mapping between PoI mechanisms and specific regulatory frameworks is provided informatively in Appendix C.

4.6 Accountability of prespecified scope

Where a prespecification attestation carries an analysis inventory, the proposition ″every enumerated analysis has a recorded outcome in this proof″ is mechanically decidable, and its violation is a verification failure at the levels where coverage applies.

This follows from §5.6: inventory entries and the `analysis_id` bindings in prespecification claim bodies are structured data; the coverage check is a finite set-mapping over them. The property is deliberately conditional — it holds exactly where an inventory exists (§1.3, fourth non-goal). Its regulatory force is that it converts the question ″did the producer suppress an analysis it committed to running?″ from a matter of trust into a matter of mechanical checking, for the prespecified scope.

Editor's note (§4). Each of these subsections currently states the property and gives a one-paragraph justification. In the published version, each will be elevated to a numbered proposition with a sketch of proof. The proofs are short — none requires more than half a page — but their correctness depends on the precise wording of §2 and §3, which is why they are deferred from this draft.

---

5. Conformance Levels

PoI defines five conformance levels: L1, L2, L3, L4A, and L4R. Each level is defined as a set of constraints on which step types must appear in a proof, which replay classes are admissible for `reason` steps that are ancestors of output steps, which roles must sign which steps, and which checks the verifier must perform.

A producer claims conformance by setting the `conformance_claim` field in the proof manifest (§2.7) to the asserted level. The manifest is signed by the producer; the conformance claim is therefore signed evidence in itself, but it is *not* an Insight Step and does not appear in the proof DAG. A verifier confirms or refutes the conformance claim by running the verification algorithm of §3 under the constraints of the claimed level.

5.0 Independence classes

Several level requirements turn on whether two attestors are ″independent.″ v0.6.2 required only that two attestor URIs resolve to distinct identities — a condition satisfied by two employees of the same sponsor, which is not what any regulatory regime means by independence. v0.7.0 grades the requirement:

- I1 — Distinct key. The attestors' resolved signing keys differ. (Weakest; defends only against single-key self-attestation.)
- I2 — Distinct individual. The attestor URIs resolve, under the profile's authority model, to distinct natural persons (or, for system attestors, to a natural person distinct from the system's operating identity).
- I3 — Distinct organization. The attestor URIs resolve to distinct legal entities under the profile's authority model.

An independence class is mechanically checkable only to the extent the profile's authority model encodes the relevant facts; a profile claiming support for I2 or I3 MUST define how individual and organizational identity, respectively, are resolved and compared. Identity distinctness at any class is necessary but not sufficient for the regulatory concept of independence (which also concerns reporting lines, compensation, and conflicts); the class system bounds what the protocol can check mechanically and leaves the rest, explicitly, to the profile's governance requirements and to human review.

5.1 Level definitions

L1 — Provenance. The proof DAG contains only steps of type `observe` and `compute`. All steps are signed. All `compute` steps declare a `replay_regime` (bit-identical or tolerance) and meet the corresponding replay requirements when verified by a party with access to the function and environment. The proof DAG contains no `reason` and no `attest` steps. At L1, `attestor` resolution MAY be limited to cryptographic key resolution sufficient to verify the signature; the bound key is not required to be linked to a verified organizational or individual identity. Profiles MAY permit self-declared or locally-issued keys at L1. Sufficient for low-risk computational analyses where no reasoning operations contribute to conclusions, no in-proof attestations are claimed as part of the evidence, and identity binding is not required by the consuming process. (External attestations, including the manifest's own signature, are unaffected by this restriction.)

L2 — Identity. L1 plus: at L2, the key resolved for every `attestor` MUST be bound, under the profile's authority model, to a verified organizational or individual identity and role, resolvable as of each step's timestamp (§3.2). All timestamps resolve to a profile-recognized timestamp authority. Sufficient for regulated computational analyses without reasoning components.

L3 — Reasoning. L2 plus: `reason` steps MAY appear. Every `reason` step that is an ancestor of an output step MUST declare `replay_class` of at least R2 — that is, R1 (recorded-only) reason steps are not permitted as ancestors of output steps at L3. The model identifier MUST be resolvable under the profile's resolution mechanism at the recorded version *at proof production time*; per §2.2.3 the resolvability may decay thereafter, and the verifier reports the achieved state. `attest` steps MAY appear and follow the rules of §2.2.4. Sufficient for agentic analyses contributing to regulated conclusions where re-execution access (but not full reproducibility) is available.

L4A — Independently Attested. L3 plus the following three requirements:

1. Independent qualified review. For every output step `s` of type `reason`, there MUST exist an `attest` step `a` in the proof such that:
   - `a` has an `about` edge pointing to `s`,
   - `a.payload.role` is in the profile-defined set of *qualified review roles*,
   - `a.payload.claim_type` is in the profile-defined set of *approval claim types*,
   - `a.attestor` and `s.attestor` satisfy independence class I2 or stronger; the profile MUST declare the required class per role (and SHOULD require I3 for roles whose regulatory function is independent validation).

   (Note that `a` is a *successor or sibling* of `s` in the DAG, not a predecessor. The relationship is established by the `about` edge from `a` to `s`, not by any edge from `s` to `a`.)

2. Prespecification for confirmatory analyses. Profiles designating a `compute` or `reason` step as a "confirmatory analysis" output MUST require that an `attest` step of `claim_type` `prespecification/locked-plan` (or a profile-specific variant) exists, with an `about` edge pointing to the analysis step, whose

`claim_body` references an external locked plan with lock evidence predating the relevant data-exposure event. The base protocol provides the structural mechanism; each profile binds the determination of "confirmatory analysis" and the data-exposure event to its domain.

Limitation (normative statement). The protocol can verify that the *attestation* and its lock evidence predate recorded events; it cannot verify that no analyst saw the data before the lock. An `observe` step's timestamp records ingestion into the proof, not first human exposure (§2.2.1). A check that compares the lock time only against observe timestamps therefore defends against *backdated prespecification attestations*, not against actual pre-exposure. Profiles MUST bind the data-exposure side of the comparison to external exposure events where their regime makes such events legible — e.g., an EDC database-lock or unblinding attestation consumed through an `observe` step's `provenance` field (§7.4) — and MUST NOT represent the observe-timestamp comparison alone as evidence of blinding integrity.

3. Coverage. Where any prespecification attestation in the effective ancestor closure carries an analysis inventory, the coverage check of §5.6 MUST pass. (This requirement is vacuous for proofs whose locked plans carry no inventory; profiles MAY require inventories within their regulated scope, at which point the requirement has force.)

L4A admits `reason` steps with `replay_class` R2 throughout, including for outputs that downstream regulatory regimes may consider high-stakes. The strength of L4A rests on the combination of identity binding (L2), re-executability with model resolution (L3, R2), independent qualified review, prespecification, and coverage rather than on bit-identical reproducibility. L4A is the level appropriate to regulated agentic workflows on closed-weight hosted models with strong governance.

L4R — Reproducible. L4A plus the following requirement:

4. Reproducibility for high-stakes reasoning. Profiles MAY designate certain output classes as "high-stakes." For any `reason` step that is an ancestor of a high-stakes output step, `replay_class` MUST be R3 (reproducible). For other output steps, R2 remains permitted. Profiles SHOULD acknowledge the aspirational status of R3 per §2.2.3 in defining their high-stakes-output set.

L4R is the strongest conformance claim and is sufficient for regulated analyses where bit-identical reproduction of high-stakes reasoning outputs is both required and achievable — characteristically open-weight model deployments under controlled runtimes. L4R subsumes L4A: a proof satisfying L4R necessarily satisfies L4A.

Choosing between L4A and L4R. Producers SHOULD claim the higher of the two levels their analysis can support. Profiles MAY restrict their regulated scope to one of the two — for example, a profile addressing a regulatory regime that demands bit-identical reproducibility for primary endpoints would require L4R; a profile addressing a regime that demands independent qualified review and prespecification but accepts non-deterministic reasoning would accept L4A. The protocol does not by itself decide which level a given regulatory regime should require; this is a profile and regulator concern.

5.2 Decision procedure

For each level, the additional checks beyond L1's are mechanically expressible as predicates over the manifest, the proof DAG, and per-step attributes. The verifier of §3 takes the level as a parameter (read from `M.conformance_claim`) and applies the corresponding predicate set. No level requires the verifier to interpret natural-language content; all checks are over types, digests, identities, replay classes, edge relations, roles, claim-body fields, and timestamps. (The coverage check reads `analysis_id` strings from structured claim bodies; it compares them for equality, it does not interpret them.)

A useful framing of the level definitions is as the following graph query: at each level, the set of output-reaching steps must satisfy a level-specific predicate set. The predicate set grows monotonically with level: L1 ⊂ L2 ⊂ L3 ⊂ L4A ⊂ L4R. An L3 verifier checking an L4A-claimed proof would not detect missing review attestations as failures; an L4A verifier checking an L1-claimed proof would not require them. Conformance is checked against the claimed level, not against the highest level the proof structurally satisfies.

5.3 The handling of **reason** divergence at L3, L4A, and L4R

At L3 and L4A, a `reason` step with `replay_class` R2 that diverges on re-execution is recorded as divergent (§3.2 reason.f) but is *not* a verification failure. The proof's conformance status is unaffected by R2 divergence at the protocol level. Profiles MAY require that divergence above a threshold trigger a supersession (§5.4) before the proof is admissible to the regulated process the profile addresses; the protocol provides the mechanism, the threshold and trigger are profile-level decisions.

At L4R, a `reason` step in the path of a high-stakes output step MUST be R3 and therefore MUST replay bit-identically. Divergence in this case IS a verification failure (§3.2 reason.f, R3 branch). This is the central reason for the R3 class: it admits the strongest, regulator-grade claim that recorded outputs can be reproduced. The split between L4A and L4R isolates this claim from the independent-attestation claims, so that regulated workflows whose reasoning is structurally non-reproducible can still earn the independent-attestation guarantees of L4A without overclaiming reproducibility.

5.4 Supersession and amendment

The protocol does not permit modification of a step once signed. Amendment is expressed by appending new steps:

- An `attest` step of `claim_type: supersession/retract` about the original step records the retraction.
- A new step (typically of the same type as the original) records the corrected derivation.
- An `attest` step of `claim_type: supersession/replace` about both the original and the replacement records the binding between them.

Verifiers MUST treat a step as superseded if there exists in the proof an `attest` step with `claim_type: supersession/retract` or `supersession/replace` about it. As specified in §3.1, the structural ancestor closure preserves the historical derivation of every output; the effective ancestor closure used for conformance evaluation excludes superseded steps. An output that depends on a superseded predecessor and is not itself superseded fails structural validation (§3.1 step 7); a producer correcting an analysis MUST therefore supersede the affected output as well, or replace it with a non-superseded successor.

This pattern preserves the immutability of the chain while admitting the practical reality that regulated analyses are revised, and prevents the inadvertent retention of conclusions whose derivation has been retracted.

5.5 Insufficient-evidence findings

A `reason` step MAY declare its `finding_type` as `no-finding`, `insufficient-evidence`, `negative-result`, or a profile-defined variant (§2.2.3). Such steps are first-class evidence: they record what the analysis examined and what it could not conclude, and they are subject to the same well-formedness, signature, timestamp, and replay requirements as any other `reason` step. They are not the absence of analysis; they are the recorded result of an analysis that did not produce a positive finding.

The protocol's treatment of `finding_type` is structural. A `finding_type` of `insufficient-evidence` does not exempt the step from any conformance check; the step's input bindings, conditioned-on relations, replay class, and (at L4A or L4R) qualified review are checked as usual. The protocol does not by itself decide whether an insufficient-evidence determination is correct; that determination is the substance of qualified review, and may be the subject of an `adequacy/finding-confirmed` or `adequacy/finding-disputed` attestation (§2.2.4).

The motivation for first-class treatment is regulatory. In pre-submission contexts and adverse-event review, the analyses that found nothing are as important as the analyses that found something. A proof that records its no-finding outcomes alongside its conclusions provides a regulator a complete view of what the analysis examined; a proof that silently omits them does not. Profiles SHOULD require explicit `finding_type` for `reason` steps in regulated contexts rather than relying on the default `conclusion`.

Until this version, however, the first-class status of no-finding evidence was an invitation, not a check: nothing made the omission of a no-finding analysis detectable. §5.6 supplies the check.

5.6 Coverage over prespecified analysis inventories

Coverage is the mechanism by which the protocol makes selective recording detectable for prespecified work. It composes two existing mechanisms — prespecification attestations (§2.2.4) and no-finding evidence (§5.5) — that were structurally adjacent but unconnected in prior versions.

Inventories. A prespecification claim body MAY carry an *analysis inventory*:

```
prespecification claim_body = {
  "plan": {
    "digest":        <digest of the locked external plan artifact>,
    "locked_at":     <RFC 3339 lock time>,
    "lock_evidence": <external trusted timestamp token over plan.digest,
                      or reference to equivalent external lock evidence>,
    "authorizers":   [ <identity references with lock-time signatures>, ... ]
  },
  "analysis_id": <string: the inventory entry this attestation's about-target
                  implements; REQUIRED when the plan carries an inventory>,
  "inventory":   <optional> [
    { "analysis_id": <string, unique within the plan>,
      "scope":       "confirmatory" | "exploratory" | <profile-defined>,
      "title_digest": <optional digest of a short human-readable description> },
    ...
  ]
}
```

The inventory travels in the claim body (and is therefore digest-bound by `claim_hash`); equivalently, a profile MAY locate the inventory inside the locked plan artifact itself, identified by `plan.digest`, in a profile-specified machine-readable region. Either way the inventory is content-addressed and signed.

The coverage check. Let *Plans* be the set of distinct `plan.digest` values appearing in prespecification attestations within the effective ancestor closure A^*. For each $p \in Plans$ whose inventory is resolvable to the verifier:

```
COVERAGE_CHECK(p):
    for each entry e in inventory(p):
        require: there exists an output step o ∈ M.outputs such that
            (i)   o is not superseded, or o is superseded and a replacement
                  for o (per supersession/replace) is itself in M.outputs
                  and not superseded; and
            (ii)  there exists a prespecification attestation a in the proof
                  with an about edge to o (or to o's replacement), with
                  a.claim_body.plan.digest = p and
                  a.claim_body.analysis_id = e.analysis_id.
        on violation: FAIL("coverage", p, e.analysis_id)
    return SATISFIED
```

Three properties of the check:

1. **Finding-type-agnostic.** An output step of any `finding_type` — `conclusion`, `no-finding`, `insufficient-evidence`, `negative-result` — satisfies coverage. That is the point: the producer discharges its prespecified obligation by recording *what happened*, including that nothing was found. Coverage and §5.5 are two halves of one mechanism.
2. **Supersession-aware.** A retracted analysis does not satisfy coverage unless replaced; a producer cannot discharge an inventory entry by recording and then retracting it (which interacts with threat 9 and 11 in §6).
3. **Conditional and honest about its limits.** If no inventory exists, coverage is vacuously satisfied and the verifier reports `coverage: not-applicable`. If an inventory exists but is not resolvable (e.g., it lives in

a plan artifact the verifier cannot retrieve), the verifier reports `coverage: not-evaluable` for that plan; profiles MAY treat not-evaluable as failure within their regulated scope, and profiles whose regime turns on multiplicity control SHOULD. The protocol does not pretend coverage says anything about work outside the inventory (§1.3).

Where coverage applies. Coverage is REQUIRED at L4A and L4R (§5.1, requirement 3) whenever an inventory is present, and verifiers MUST evaluate and report it at every level whenever an inventory is present. Profiles bind whether inventories themselves are required: a profile whose regulated scope includes confirmatory analyses SHOULD require that the locked plan enumerate them.

Editor's note (§5). The level definitions are committed; the precise *role vocabularies*, *approval claim-type vocabularies*, *high-stakes output classifications*, *confirmatory-analysis designations*, and *required independence classes* that L4A and L4R require are profile-specific and not specified here. Reference vocabularies for the clinical-trials and finance regimes are sketched in §7.4 and §7.5 respectively.

---

6. Threat Model

Editor's note (§6). This section remains an enumerated placeholder pending stabilization of §2–§3 and §5. The intended contents:

The threat model section will enumerate, for each of the following adversaries, what PoI provides and what it does not:

1. **Tampering producer.** A producer attempting to alter a proof after the fact. (Defended by hash chaining, signatures, and timestamping; see §4.4 for the precise statement, including the timestamp-substitution residual.)
2. **Compromised attestor key.** A producer or external party in possession of an attestor's signing key. (Not defended at the protocol level; identity revocation is a profile concern. Signing-time authorization evaluation (§3.2) surfaces the compromise window; profiles define remediation for steps timestamped within it.)
3. **Colluding attestors.** Multiple attestors cooperating to produce a fraudulent but internally consistent proof. (Not defended at the protocol level for L1–L3; partially mitigated at L4A and L4R by independence-class requirements (§5.0) — note that I3 raises the collusion cost from intra-organizational to inter-organizational, it does not eliminate it.)
4. **Source-data fabrication.** An attestor presenting fabricated data as an `observe`. (Out of scope per §1.3 and §2.2.1; delegated to source-data integrity standards.)
5. **Model substitution.** A `reason` step whose recorded model identifier does not correspond to the model actually used. (Defended where weights digests are recorded and the profile permits resolution of the digest; not defended for hosted models identified only by version string — the producer's claim is then attested, not verified.)
6. **Replay-time model drift.** A model whose weights have changed between original execution and verification. (Defended where weights are digested; not defended where weights are referenced only by version string. See also "R2 claims decay," §2.2.3.)
7. **Timestamp authority compromise.** A timestamp authority issuing tokens for backdated content. (Defended only to the extent the profile's authority is itself trustworthy; PoI inherits the trust assumptions of its timestamp profile. Transparency-log authorities reduce this to a publicly auditable failure.)
8. **Post-hoc prespecification fabrication.** A producer constructing a "prespecification" attestation after observing the data. (Defended where the lock evidence — the external trusted timestamp over the plan digest — predates the relevant *external* data-exposure event, and the timestamp authority is itself trustworthy. The §5.1 limitation statement applies: the protocol verifies the ordering of recorded events, not the absence of unrecorded exposure.)
9. **Silent ancestor retraction.** A producer retracting a predecessor without retracting the output that depends on it, attempting to retain a conclusion whose derivation has been disavowed. (Defended by §3.1 step 7.)

10. **Verification-basis overclaiming.** A producer claiming `verification_basis: replay-verifiable` for a proof whose `reason` steps are not actually re-executable to verifiers, or declaring `archival-complete` for a bundle with gaps. (Detected: the verifier independently confirms both the achieved basis and the bundle completeness, §2.8 and §3.5.)
11. **Selective recording (file-drawer).** A producer omitting from the proof analyses whose results were unfavorable. (Defended, for prespecified work, by the coverage check of §5.6: every inventoried analysis must map to a recorded outcome or explicit supersession. Not defended for work outside any inventory — stated as a non-goal in §1.3. The defense is therefore exactly as strong as the prespecification culture of the regime applying it, which is the correct division of labor between a protocol and a regulator.)
12. **Input-channel manipulation.** Content injected into the inputs of a `reason` step — characteristically through tool responses during inference (prompt injection in retrieved material) — steering the recorded conclusion. (Not defended: the protocol records the channel, it cannot police it. Mitigated for *review* by the tool-call log carrier and, in regulated profiles, the expanded per-call representation (§2.2.3), which make the channel inspectable; an injected instruction visible in a hash-bound tool log is discoverable by a reviewer in a way that an unlogged one never is.)
13. **Redaction abuse.** A producer using disclosure-limited carriage (§2.2.5) to conceal material content from reviewers — redacting more than the registered policy permits, or redacting content whose disclosure would change the review outcome. (Not mechanically defended for unauthorized verifiers, by construction; mitigated by (a) the binding digest, which makes the unredacted content fixed and producible on demand to authorized verifiers, (b) the registered-policy requirement and the `qualification/redaction-applied` attestation, which make the redaction an identifiable person's signed act, and (c) the authorized-verifier tier, under which a regulator with access authority verifies the binding digest directly.)

The section will end with an explicit list of *attacks PoI does not defend against*, mirroring the non-goals of §1.3 in adversarial form.

---

7. Profiles (Informative)

7.0 The Profile Mechanism

A PoI *profile* is a specification document that binds the abstract protocol of §§2–3 to a concrete cryptographic ecosystem, identity model, and predicate vocabulary. Profiles are the protocol's mechanism for domain and ecosystem specialization without revising the base protocol.

The profile mechanism specified in this subsection is normative. The individual profile sketches in §7.1–§7.5 are informative placeholders; they become normative only if separately published as formal profile specifications.

A profile MAY specify:

- The signature scheme and key infrastructure (e.g., Sigstore keyless via OIDC), including the registered signature-object `alg` identifiers (§1.5)
- The digest algorithms registered beyond `sha-256` (§1.5)
- The timestamp authority and its verification procedure (e.g., Rekor inclusion proofs, RFC 3161 tokens), and the skew tolerance δ (§2.4)
- The canonical encoding algorithm if other than JCS, and any registered canonical artifact encodings beyond `jcs+json` and `octet-stream` (§2.5.1)
- The function-identifier resolution mechanism for `compute` steps
- The model-identifier and weights resolution mechanism for `reason` steps, and the expected review horizon against which R2 decay (§2.2.3) and archival-completeness requirements (§2.8) are set
- The resolution mechanism for content-addressed references, and the offline trust snapshot format for bundle verification (§2.8)
- The registered vocabulary of claim-type identifiers, the base URI for compact names (§2.2.4), and the authorized roles for each

- The historical-authority resolution mechanism supporting signing-time authorization checks (§3.2)
- The required independence class per review and validation role (§5.0, §5.1)
- Whether the extended form of `conditioned-on` edges (§2.3) is required, and if so the registered `context_role` vocabulary
- The interpretation of "high-stakes output" and "confirmatory analysis" at L4A and L4R (§5.1)
- Whether analysis inventories (§5.6) are required for the profile's confirmatory scope, and whether `coverage: not-evaluable` is treated as failure
- The registered redaction policies, the roles authorized to apply each, whether a `qualification/redaction-applied` attestation is required, and the access model defining authorized verifiers (§2.2.5)
- Whether disclosure-limited artifacts are admissible at each conformance level within the profile's regulated scope
- Whether expanded tool-call representation (§2.2.3) is required for designated `reason` steps
- The required `verification_basis` (§2.7) and bundle completeness (§2.8) for proofs within the profile's regulated scope
- The equivalence predicate vocabulary for tolerance-regime `compute` replay
- The `finding_type` vocabulary beyond the base set (§5.5)
- The input-message reconstruction predicate for `reason` step verification (§3.2 reason.d)
- Domain-specific predicates expressed as additional `attest` claim types, including multi-replay stability evidence (§3.3)
- Whether output steps may be of types other than `compute` and `reason` (§2.7)

A profile MUST NOT modify the four-step taxonomy, the three edge relations, or the verification algorithm of §3. Extension is by predicate vocabulary and resolution mechanism, not by new step types or new verification semantics. This constraint preserves the property that a single verifier implementation can verify a proof under any profile, given the profile's resolution services.

A proof manifest's `profiles` field (§2.7) lists the profile(s) under which the proof is to be verified; the verifier MUST apply each named profile's resolution rules and predicate set in addition to the base protocol's requirements.

7.1 Sigstore profile

Binds signature to Sigstore's keyless signing scheme. Binds timestamp to Rekor inclusion proofs (which additionally provide the transparency-log properties noted in §2.1 and §6 threat 7). Specifies attestor URI form using OIDC subject identifiers. Specifies the authority model for §3.2's `observe` and `attest` checks via OIDC claim verification, including historical resolution against the log for signing-time authorization.

7.2 in-toto interoperability profile

Specifies how an `observe` step's `provenance` field references an in-toto attestation, and how a verifier consumes the in-toto attestation as part of the `observe` check. Provides a reverse mapping: a sequence of in-toto attestations representing a build pipeline can be lifted into an L1 PoI proof.

7.3 RFC 3161 timestamping profile

Specifies token format and authority resolution for environments where Sigstore Rekor is not appropriate, including the binding of the token to the step identity (§2.4) and the offline verification procedure for archived tokens within bundles.

7.4 Clinical Trials Profile (Informative, Forward-Reference)

A profile targeting clinical-trial readiness analyses and AI-assisted clinical decision support. Anticipated specializations include:

- **Identity model.** Industry-standard signing scheme (e.g., Sigstore via institutional OIDC) with role bindings for `principal-investigator`, `biostatistician`, `qualified-reviewer`, `independent-validator`, `sponsor-medical-monitor`, and `data-monitoring-committee-member`.

- **Independence classes.** `independent-validator` and `data-monitoring-committee-member` at I3 (distinct organization, §5.0); `qualified-reviewer` at I2 minimum with profile-level governance requirements (reporting-line separation) stated as non-mechanical obligations.
- **Source-data integrity bindings.** `observe` step `provenance` references MUST resolve to attestations from systems operating under 21 CFR Part 11 §11.10 (clinical data management systems with audit trails meeting predicate-regulation requirements), or to equivalent attestations under EU CTR Annex 11 or ICH E6(R3) §5.5.
- **Unblinding bound to external events, not observe timestamps.** Per the §5.1 limitation statement: the data-exposure side of every prespecification comparison MUST be an externally attested exposure event — characteristically an EDC database-lock or unblinding attestation consumed through an `observe` step's `provenance` field — not the observe step's own ingestion timestamp. The profile defines the recognized exposure-event attestation forms.
- **Prespecification predicates.** Required claim-type vocabulary including `prespecification/locked-sap`, `prespecification/locked-protocol`, `prespecification/locked-charter`, each binding to a pre-DBL content-digested external artifact with external lock evidence.
- **Analysis inventories required for confirmatory scope.** Locked SAPs within the profile's confirmatory scope MUST enumerate an analysis inventory (§5.6) covering primary, secondary, and prespecified subgroup and sensitivity analyses; `coverage: not-evaluable` is treated as failure within that scope. This is the profile-level binding that gives the protocol's coverage mechanism its ICH E9(R1) multiplicity-control force.
- **Subgroup analysis predicates.** Claim-type vocabulary including `subgroup-analysis/prespecified` and `subgroup-analysis/post-hoc`, allowing structural distinction between confirmatory and exploratory subgroup findings.
- **Insufficient-evidence handling.** `finding_type` vocabulary aligned with the ICH E9(R1) estimand framework, distinguishing intercurrent-event treatment and missingness handling. Required for `reason` steps contributing to safety or efficacy determinations.
- **Expanded tool-call representation.** REQUIRED for `reason` steps that are ancestors of safety or efficacy outputs (§2.2.3): each tool call recorded as a `compute` substep with explicit edges.
- **PHI redaction.** Disclosure-limited carriage (§2.2.5) REQUIRED for artifacts containing protected health information, under a registered de-identification policy, with a `qualification/redaction-applied` attestation by an authorized role; the regulator is the canonical authorized verifier.
- **Bias and fairness predicates.** Claim-type vocabulary for `bias-assessment/subgroup-performance`, `bias-assessment/representativeness`, supporting structural distinction between assessed and unassessed bias dimensions.
- **Conformance level for hosted-model workflows.** Anticipated to accept L4A for analyses on closed-weight hosted models with the required independent attestations, prespecification, coverage, and identity governance; L4R required for analyses where bit-identical reproducibility of primary endpoint reasoning is achievable and demanded by the regulatory regime.
- **High-stakes output classification at L4R.** Any `reason` or `compute` step contributing to a primary or co-primary efficacy endpoint, or to a safety determination at the protocol's prespecified threshold.
- **Submission packaging.** Proofs submitted within the profile's regulated scope are carried as archival-complete bundles (§2.8); the profile specifies the offline trust snapshot and the placement of bundles within the applicable submission structure (e.g., eCTD module placement), which is a profile concern, not a base-protocol concern.

7.5 Finance Profile (Informative, Forward-Reference)

A profile targeting model risk management under SR 11-7 and analogous regulatory regimes. Anticipated specializations include:

- **Identity model.** Institutional signing with role bindings for `model-developer`, `model-validator` (independent per SR 11-7), `model-owner`, `model-risk-officer`.
- **Independence classes.** `model-validator` at I2 minimum with the firm's organizational-separation requirements stated as governance obligations; I3 where validation is outsourced.
- **Source-data integrity bindings.** `observe` step `provenance` MUST resolve to attestations from systems

operating under firm-level books-and-records controls.
- **Tiering predicates.** Claim-type vocabulary aligned with the firm's model tiering (typically three or four tiers); high-stakes classification under L4R at this profile is mapped to the firm's tier-1 or tier-2 designations.
- **Effective challenge predicates.** Claim-type vocabulary for `model-validator` attestations recording independent replication, conceptual soundness review, and ongoing-monitoring sign-off; inventories (§5.6) over validation charters give the challenge program's planned scope mechanical coverage.
- **Lifecycle predicates.** Claim-type vocabulary for `production-deployment`, `periodic-revalidation`, `retirement`, supporting the lifecycle obligations of SR 11-7.

Profile sketches §7.1–§7.5 are informative with respect to the base protocol — non-normative until separately published as formal profile specifications — but a producer MUST identify the profile(s) under which a proof is to be verified, by reference, in the proof manifest (§2.7).

---

8. Related Work

Editor's note (§8). This section will distinguish PoI from the most directly comparable prior work. Planned coverage:

- **in-toto and SLSA.** The closest cryptographic substrate. PoI's signature, content-addressing, and timestamping primitives are intended to be instantiated via Sigstore and Rekor under the §7.1 profile; `observe` step provenance is the explicit integration point with in-toto attestations under the §7.2 profile. PoI's contribution beyond these frameworks is the typed-step taxonomy admitting non-deterministic `reason` steps, the typed edge relations, the visible-rationale evidence field, and the verification algorithm extending replay semantics to non-deterministic operations. PoI does not replace in-toto/SLSA; it extends their attestation lineage into operations they were not designed to address.
- **Certificate Transparency (RFC 6962).** Closest precedent for a protocol whose verification algorithm is mechanically tractable and whose claims are precisely scoped. PoI borrows CT's discipline of stating exactly what the protocol does and does not assert.
- **PROV-W3C and PROV-AGENT.** Closest data-model precedents for agent-aware provenance. PoI differs in that it is a verification protocol with a single mechanical algorithm, not a data model. PoI proofs can be lifted to PROV representations for interoperability, but PROV does not, by itself, give a verifier.
- **Ojewale et al., "Audit Trails for Generative-AI Workflows" (FAccT 2026).** Closest contemporary academic treatment of this problem. PoI's contribution beyond their framework is the explicit four-type taxonomy, the typed edge relations, the visible-rationale field as first-class evidence, the coverage mechanism over prespecified inventories, and the verification algorithm with stated complexity bound.
- **ASME V&V40 and the FDA AI/ML draft guidance.** The most directly relevant regulatory framings. PoI does not implement either; it is structured so that compliance rule packs corresponding to either can be expressed as predicates over the proof. The mapping in Appendix C is informative.
- **ProvCaRe and clinical research provenance.** Closest prior work in clinical-trial provenance specifically; PoI's clinical profile (§7.4) is intended to interoperate with ProvCaRe's predicate vocabulary where applicable.

Appendix A. JSON Schema for the Insight Step (Illustrative)

Editor's note. This appendix gives a JSON Schema (Draft 2020-12) for the Insight Step. The schema is *illustrative*, not normative; the normative definition is the prose of §2. JSON Schema validation is the weakest of five distinct validity gates a proof must pass. This version regenerates the schema against the v0.7.0 field set: digests and signatures are structured objects (§1.5), artifact-bearing fields use the disclosure-aware carrier

of §2.2.5, `output_encoding` is REQUIRED on `compute` and `reason`, and `claim_type` admits both absolute URIs and compact `family/name` identifiers (§2.2.4).

The five gates, in order of increasing strength:

1. **Schema-valid.** The step's JSON encoding satisfies the schema below. This catches typos, malformed fields, missing required keys, and most type errors. It does not check hash linkage, signature validity, role authorization, topological ordering, acyclicity, edge uniqueness, or conformance-level predicates.
2. **Structurally valid.** The step (or proof of steps) satisfies §2.6 and, at the proof level, §3.1. This adds acyclicity, predecessor existence, the prohibition on duplicate edges to the same predecessor, timestamp ordering within skew tolerance (§2.4), the manifest-to-proof correspondence, and the prohibition on `attest-as-derived-from`.
3. **Cryptographically valid.** Signatures over `to_sign` verify; the step identity equals `digest(to_timestamp)` — fields 1–6, excluding the timestamp (§2.1, §2.5); and the timestamp token verifies over that identity (§2.4). This adds the integrity guarantees of the underlying cryptographic primitives.
4. **Type-verification-valid.** §3.2's type-specific checks pass for every step. For `compute` and `reason`, this includes invocation reconstruction and (where applicable) replay; for disclosure-limited artifacts, the achievable basis depends on the verifier's authorization tier (§2.2.5).
5. **Conformance-valid.** §5's level-specific predicates hold for the manifest's claimed level, evaluated over the effective ancestor closure A^* (§3.1), including coverage over any prespecified analysis inventory (§5.6).

A claim of "this proof is valid" must specify which gate is meant. Most commonly the relevant claim is gate 5 (conformance-valid against a specific claimed level under a specific profile). The schema below is gate 1. It does not enforce conditional requirements stated in prose (for example, `output_artifact` required at R1 or under `tolerance`, `weights_hash` required at R3, edge uniqueness, or consistency of the `redactions` record with the carriers used); these are checked at higher gates. The `verification_basis` field (§2.7) reports producer claims about which gates the proof is expected to reach.

Scope of this appendix. The schema below covers the Insight Step only. The manifest (§2.7), manifest-level attestation (§2.9), archival bundle (§2.8), and verification report (§3.5) are equally machine-checkable, but their schemas are published and version-locked with the conformance test suite (Appendix B), where they are exercised against test vectors rather than published untested here — keeping every published schema a tested one.

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://proofofinsight.org/schema/v0.7.0/step.json",
  "title": "PoI Insight Step",
  "type": "object",
  "additionalProperties": false,
  "required": ["version", "type", "predecessors", "payload", "attestor", "signature", "timestamp"],
  "properties": {
    "version":   { "const": "0.7.0" },
    "type":      { "enum": ["observe", "compute", "reason", "attest"] },
    "predecessors": {
      "type": "array",
      "items": {
        "oneOf": [
          {
            "type": "object",
            "additionalProperties": false,
            "required": ["step", "relation"],
            "properties": {
              "step":     { "$ref": "#/$defs/digest" },
              "relation": { "enum": ["derived-from", "conditioned-on", "about"] }
```

```
          }
        },
        {
          "type": "object",
          "additionalProperties": false,
          "required": ["step", "relation", "context_role", "declared_relevance_hash"],
          "properties": {
            "step":                  { "$ref": "#/$defs/digest" },
            "relation":              { "const": "conditioned-on" },
            "context_role":          { "type": "string" },
            "declared_relevance_hash": { "$ref": "#/$defs/digest" }
          }
        }
      ]
    }
  },
  "payload":   { "type": "object" },
  "attestor":  { "type": "string", "format": "uri" },
  "signature": { "$ref": "#/$defs/signatureObject" },
  "timestamp": {
    "type": "object",
    "additionalProperties": false,
    "required": ["value", "authority", "token"],
    "properties": {
      "value":     { "type": "string", "format": "date-time" },
      "authority": { "type": "string", "format": "uri" },
      "token":     { "type": "string", "minLength": 1 }
    }
  }
},
"$defs": {
  "digest": {
    "type": "object",
    "additionalProperties": false,
    "required": ["alg", "value"],
    "properties": {
      "alg":   { "type": "string", "pattern": "^[a-z0-9][a-z0-9-]*$" },
      "value": { "type": "string", "pattern": "^([0-9a-f]{2})+$" }
    }
  },
  "signatureObject": {
    "type": "object",
    "additionalProperties": false,
    "required": ["alg", "value"],
    "properties": {
      "alg":    { "type": "string", "minLength": 1 },
      "key_id": { "type": "string" },
      "value":  { "type": "string", "pattern": "^[A-Za-z0-9+/]+={0,2}$" }
    }
  },
  "contentRef": {
    "type": "object",
    "additionalProperties": false,
    "required": ["uri", "digest"],
```

```
      "properties": {
        "uri":    { "type": "string", "format": "uri" },
        "digest": { "$ref": "#/$defs/digest" }
      }
    },
    "inlineOrRef": {
      "anyOf": [
        { "type": "object" },
        { "$ref": "#/$defs/contentRef" }
      ]
    },
    "disclosureLimited": {
      "type": "object",
      "additionalProperties": false,
      "required": ["binding_digest", "disclosed", "disclosed_digest", "policy"],
      "properties": {
        "binding_digest": { "$ref": "#/$defs/digest" },
        "disclosed": {
          "anyOf": [
            { "type": ["object", "array", "string"] },
            { "$ref": "#/$defs/contentRef" }
          ]
        },
        "disclosed_digest": { "$ref": "#/$defs/digest" },
        "policy": {
          "anyOf": [
            { "type": "string" },
            { "$ref": "#/$defs/contentRef" }
          ]
        }
      }
    },
    "artifactCarrier": {
      "anyOf": [
        { "type": ["object", "array", "string"] },
        { "$ref": "#/$defs/contentRef" },
        { "$ref": "#/$defs/disclosureLimited" }
      ]
    },
    "outputEncoding": {
      "anyOf": [
        { "enum": ["jcs+json", "octet-stream"] },
        { "type": "string", "minLength": 1 }
      ]
    }
  },
  "allOf": [
    {
      "if":   { "properties": { "type": { "const": "observe" } } },
      "then": {
        "properties": {
          "predecessors": { "maxItems": 0 },
          "payload": {
            "type": "object",
```

```
        "additionalProperties": false,
        "required": ["content_hash", "content_type", "source"],
        "properties": {
          "content_hash": { "$ref": "#/$defs/digest" },
          "content_type": { "type": "string" },
          "source":       { "type": ["string", "object"] },
          "provenance":   { "type": ["string", "object"] }
        }
      }
    }
  }
},
{
  "if":   { "properties": { "type": { "const": "compute" } } },
  "then": {
    "properties": {
      "predecessors": {
        "minItems": 1,
        "items": { "properties": { "relation": { "const": "derived-from" } } }
      },
      "payload": {
        "type": "object",
        "additionalProperties": false,
        "required": ["function", "invocation", "invocation_hash", "output_encoding", "output_hash", "enviro
        "properties": {
          "function":        { "type": "string" },
          "invocation":      { "$ref": "#/$defs/inlineOrRef" },
          "invocation_hash": { "$ref": "#/$defs/digest" },
          "output_encoding": { "$ref": "#/$defs/outputEncoding" },
          "output_hash":     { "$ref": "#/$defs/digest" },
          "output_artifact": { "$ref": "#/$defs/artifactCarrier" },
          "environment": {
            "type": "object",
            "required": ["replay_regime"],
            "properties": {
              "replay_regime": { "enum": ["bit-identical", "tolerance"] }
            }
          }
        }
      }
    }
  }
},
{
  "if":   { "properties": { "type": { "const": "reason" } } },
  "then": {
    "properties": {
      "predecessors": {
        "minItems": 1,
        "items": {
          "properties": {
            "relation": { "enum": ["derived-from", "conditioned-on"] }
          }
        }
```

```
      },
      "payload": {
        "type": "object",
        "additionalProperties": false,
        "required": [
          "model", "replay_class",
          "invocation", "invocation_hash",
          "input_messages", "input_messages_hash",
          "output_encoding", "output_hash", "sampling"
        ],
        "properties": {
          "model": {
            "type": "object",
            "additionalProperties": false,
            "required": ["identifier"],
            "properties": {
              "identifier":   { "type": "string" },
              "weights_hash": { "$ref": "#/$defs/digest" },
              "version":      { "type": "string" }
            }
          },
          "replay_class":           { "enum": ["R1", "R2", "R3"] },
          "invocation":             { "$ref": "#/$defs/inlineOrRef" },
          "invocation_hash":        { "$ref": "#/$defs/digest" },
          "input_messages":         { "$ref": "#/$defs/artifactCarrier" },
          "input_messages_hash":    { "$ref": "#/$defs/digest" },
          "tool_call_log":          { "$ref": "#/$defs/artifactCarrier" },
          "tool_call_log_hash":     { "$ref": "#/$defs/digest" },
          "visible_rationale":      { "$ref": "#/$defs/artifactCarrier" },
          "visible_rationale_hash": { "$ref": "#/$defs/digest" },
          "finding_type": {
            "anyOf": [
              { "enum": ["conclusion", "no-finding", "insufficient-evidence", "negative-result"] },
              { "type": "string", "pattern": "^[a-z][a-z0-9\\-/]*$" }
            ]
          },
          "output_encoding":        { "$ref": "#/$defs/outputEncoding" },
          "output_hash":            { "$ref": "#/$defs/digest" },
          "output_artifact":        { "$ref": "#/$defs/artifactCarrier" },
          "sampling":               { "type": "object" },
          "redactions":             { "type": ["object", "array"] }
        }
      }
    }
  },
  {
    "if":   { "properties": { "type": { "const": "attest" } } },
    "then": {
      "properties": {
        "predecessors": {
          "minItems": 1,
          "items": { "properties": { "relation": { "const": "about" } } }
        },
```

```
        "payload": {
          "type": "object",
          "additionalProperties": false,
          "required": ["claim_type", "role", "claim_body", "claim_hash"],
          "properties": {
            "claim_type": {
              "anyOf": [
                { "type": "string", "format": "uri", "pattern": "^[a-z][a-z0-9+.-]*:" },
                { "type": "string", "pattern": "^[a-z][a-z0-9-]*/[a-z0-9-]+$" }
              ]
            },
            "role":       { "type": "string" },
            "claim_body": { "type": ["object", "string"] },
            "claim_hash": { "$ref": "#/$defs/digest" }
          }
        }
      }
    }
  ]
}
```

Appendix B. Conformance Test Suite (Forward Reference, Elevated Priority)

Editor's note. A companion document, the *PoI Conformance Test Suite v0.7.0*, will provide a set of concrete proof DAGs labeled with their expected verification outcome at each conformance level, together with the regenerated machine-readable schemas. The test suite is the operational specification of conformance: an implementation is conformant iff it agrees with the test suite on every test case.

This version elevates the test suite's priority explicitly: the test suite and the Core Test Profile are the next deliverables, ahead of the §4 proposition proofs and ahead of formalizing the regulatory profiles. Three reasons. First, the patent covenant (§0.3) is structured around the test suite; its absence keeps the covenant in interim language. Second, every change in this version that has bitten in review — canonical encoding of artifacts, skew handling, duplicate edges, claim-type identifier form — is exactly the class of ambiguity that executable test vectors expose and prose review does not; the cheapest time to find the remaining ones is before any implementation exists to be wrong. Third, the regulatory profiles (§7.4, §7.5) cannot responsibly bind a base protocol that has never been executed end-to-end.

The *PoI Core Test Profile* — a minimal executable profile providing concrete bindings (Ed25519 signatures, RFC 3161 or mock timestamps, inline-only invocations and artifacts, `sha-256` throughout, deterministic toy compute functions, fixed claim-type vocabulary, a trivial two-entry analysis inventory exercising §5.6, a mock redaction policy exercising §2.2.5, and no external model replay) — is the vehicle: it makes §3 and §5 testable end-to-end without depending on production identity or model-resolution infrastructure. New v0.7.0 test obligations: coverage satisfaction/violation/not-evaluable vectors; disclosure-limited verification at both authorization tiers; bundle completeness confirmation including a false `archival-complete` declaration; skew-boundary timestamp vectors; identity vectors confirming timestamp exclusion; and manifest-level attestation subject-mismatch rejection.

Appendix C. Regulatory Mapping (Informative)

This appendix maps PoI features to commonly referenced regulatory and standards frameworks. The mapping is informative: PoI does not implement any specific regulation, and conformance to PoI does not by itself establish compliance with any regulation. The mapping is provided to facilitate use of PoI as evidence within compliance regimes.

C.1 21 CFR Part 11 (Electronic Records, Electronic Signatures)

Requirement	PoI mechanism
§11.10(a) Validation of systems	Not addressed; system validation is upstream of PoI
§11.10(b) Generation of accurate and complete copies	Supports fidelity checking for recorded artifacts via content-addres
§11.10(c) Protection of records	Supports tamper-evident protection of records (§4.4); operational pr
§11.10(e) Secure, computer-generated audit trails	Provides a hash-chained typed-step audit trail with signatures and
§11.10(g) Authority checks	Supports profile-defined attestor role authorization, evaluated as of
§11.50 Signature manifestations	Provides signed `attest` steps with attestor role recorded (§2.2.4); dis
§11.70 Signature/record linking	Cryptographically links signature to record via canonical encoding

C.2 SR 11-7 (Model Risk Management)

Principle	PoI mechanism
Model development documentation	Supports documentation through `compute` and `reason` steps with full invocation hash
Effective challenge	Supports the structural form via L4A/L4R independent qualified review with graded
Ongoing monitoring	Supports lifecycle predicates defined in §7.5 finance profile
Outcomes analysis	Supports `reason` steps with insufficient-evidence findings (§5.5) and coverage over va
Model inventory	Supports manifest registry across proofs (profile-level)

C.3 FDA AI/ML Draft Guidance (Seven-Step Credibility Framework)

Step	PoI mechanism
Define the question of interest	Supports binding via prespecification attestations to study protocol (§7.4)
Define the context of use	Supports profile-level "high-stakes output" classification (§5.1)
Assess model risk	Supports producer-level threat-model documentation (§6); risk tiering is profile-defined
Develop the model	Supports model development records via `compute`/`reason` steps with invocation hashing
Verify and validate the model	Supports verification via replay regimes (§2.2.2, §2.2.3) and L4A/L4R independent validatio
Document the model	The proof itself, manifest, bundle (§2.8), and verification report (§3.5) provide the documen
Maintain the model	Supports maintenance via supersession patterns (§5.4), lifecycle predicates, and explicit R2

C.4 NIST AI Risk Management Framework (AI RMF 1.0)

Function	PoI mechanism
GOVERN	Supports governance via conformance levels, manifest signing, manifest-level attestations (§2.9), and editorial
MAP	Supports mapping via profile binding and attestor role definitions
MEASURE	Supports measurement via replay regimes, divergence semantics (§3.3), insufficient-evidence findings (§5.5), a
MANAGE	Supports management via supersession (§5.4) and lifecycle predicates in profiles

C.5 ISO/IEC 42001 (AI Management System)

Clause area	PoI mechanism
AI system lifecycle records	Supports lifecycle records via hash-chained proof DAG and archival bundles
Data quality controls	Supports referencing of upstream data quality via `observe` step provenance (delegated per §2.2
Validation and verification	Supports validation/verification via replay regimes, `attest` claim types, and the verification re
Continual improvement	Supports continual improvement via supersession and lifecycle predicates in profiles

C.6 ICH E6(R3) (Good Clinical Practice) and ICH E9(R1) (Statistical Principles)

Principle	PoI mechanism
Quality by design	Supports quality-by-design via prespecification predicates (§7.4)
Risk-based approach	Supports risk-based scoping via profile-level high-stakes classification at L4A/L4R
Data integrity (ALCOA+)	Supports ALCOA+ properties via tamper-evidence, attestor identification, and times
Investigator oversight	Supports oversight via qualified-reviewer attestations at L4A/L4R with declared inde
Estimand framework	Supports estimand-aligned `finding_type` vocabulary in clinical profile (§7.4)
Multiplicity and selective reporting	Supports mechanical coverage checking over prespecified analysis inventories (§5.6):

C.7 What These Mappings Do Not Establish

Each row above identifies a structural correspondence between a PoI mechanism and a regulatory requirement. None establishes that a proof conforming to PoI at any level constitutes compliance with the regulation. Compliance is determined by the regulator under the regulation's own evidentiary standards; PoI's role is to provide structured, mechanically verifiable evidence against which the producer's compliance claim can be assessed. PoI conformance is a necessary structural property for mechanical verification, not a sufficient compliance determination. The verification-versus-credibility distinction of §1.4 applies in full to every mapping in this appendix.

References

Editor's note. Reference list to be expanded. Anchor references already identified:

- RFC 2119, RFC 8174 — requirement keyword conventions.
- RFC 3161 — Time-Stamp Protocol.
- RFC 3339 — Date and Time on the Internet.
- RFC 6962 — Certificate Transparency.
- RFC 8785 — JSON Canonicalization Scheme.
- in-toto specification, latest version.
- SLSA specification, v1.0 or current.
- Sigstore project documentation, latest version.
- PROV-W3C, PROV-DM: The PROV Data Model.
- Souza et al., PROV-AGENT (full citation pending).
- Ojewale et al., *Audit Trails for Generative-AI Workflows*, FAccT 2026 (full citation pending).
- FDA, *Artificial Intelligence and Machine Learning in Software as a Medical Device*, draft guidance.
- FDA, *Considerations for the Use of Artificial Intelligence to Support Regulatory Decision-Making for Drug and Biological Products*, draft guidance.
- ASME V&V40, *Assessing Credibility of Computational Modeling through Verification and Validation*.
- ICH E6(R3) — Good Clinical Practice.
- ICH E9(R1) — Statistical Principles for Clinical Trials, Addendum on Estimands.
- 21 CFR Part 11.
- Federal Reserve SR 11-7, *Guidance on Model Risk Management*.
- NIST AI Risk Management Framework (AI RMF) 1.0.
- ISO/IEC 42001 — Information technology — Artificial intelligence — Management system.
- ProvCaRe project documentation (clinical research provenance).

Appendix D. Changes from v0.6.2 (Informative)

This appendix summarizes the substantive changes in v0.7.0 relative to v0.6.2, grouped by compatibility impact. Unlike the v0.6.1→v0.6.2 transition, this revision is not backward-compatible: the identity rule and value encodings change, so every v0.6.2 proof must be re-issued under v0.7.0 to claim v0.7.0 conformance. No production deployments exist; the impact population is reviewers and the (not yet released) reference implementation.

D.1 Breaking changes

- **Identity rule** (§2.1, §2.5). Step identity is now `digest(to_timestamp(step))` — fields 1–6, excluding the timestamp — and the timestamp authority attests to the identity directly. Motivation: removes TSA-latency serialization of proof construction, makes identity stable under token nondeterminism, and permits re-timestamping under authority migration. The trade-off (timestamp no longer pinned by descendants' hashes) is analyzed in §2.1 and §4.4.
- **Structured digest objects** (§1.5). Every digest is `{alg, value}`, pulled forward from the v0.6.2 deferral list. This is the hash-agility mechanism; taking it in the same revision as the identity change means identity-affecting encoding changes happen exactly once before 1.0.
- **Structured signature objects** (§1.5). Every signature is `{alg, key_id?, value}`, likewise pulled forward.
- **`output_encoding`** REQUIRED (§2.2.2, §2.2.3, §2.5.1). Compute and reason payloads must declare the canonical artifact encoding governing their output digest.
- **Duplicate-edge prohibition** (§2.3, §2.6). A step may not reference the same predecessor through more than one edge. (Tightening; any v0.6.2 proof exercising the ambiguity was relying on undefined behavior.)
- **`redaction_policy`** removed (§2.2.3). Replaced by the disclosure-limited carrier and `redactions` record of §2.2.5.
- **Version constants.** `"0.7.0"` in steps, manifests, attestations, bundles, reports.

D.2 Additive mechanisms

- **Coverage over prespecified analysis inventories** (§5.6, §5.1 L4A req. 3, §4.6, threat 11). Prespecification claim bodies may carry an analysis inventory; where one exists, every entry must map to a non-superseded output (of any `finding_type`) bound by an `analysis_id`. Makes selective recording mechanically detectable for prespecified work. Vacuous where no inventory exists, so v0.6.2-shaped proofs migrate without new obligations until a profile requires inventories.
- **Disclosure-limited artifacts** (§2.2.5, threat 13). Dual-commitment redaction: binding digest over unredacted content, disclosed redacted form, registered policy, authorized-verifier tier, per-step basis accounting.
- **Archival bundles** (§2.8, threat 10 extension). Self-contained packaging with independently confirmed completeness; offline verification through gate 4; RECOMMENDED interchange format for regulated submission. Companion normative text on R2 decay (§2.2.3).
- **Verification report format** (§3.5). The previously carrier-less reporting obligation now has a normative JSON form, making verifier outputs interoperable evidence.
- **Manifest-level attestations** (§2.9). Specified as separately-signed sibling objects, completing the v0.6.2 deferral.
- **Tool-call log and visible-rationale carriers** (§2.2.3). The artifacts behind the existing hashes gain defined homes in the payload.
- **Independence classes I1/I2/I3** (§5.0). L4A requirement 1 strengthened from "distinct identity" to ≥I2 with profile-declared class per role.
- **Skew tolerance δ** (§2.4, §3.1). Timestamp ordering checked with profile-set tolerance (default 300 s), with the ordering guarantee correctly attributed to the hash chain.
- **Signing-time authorization** (§3 preamble, §3.2). All authority checks evaluate as of `timestamp.value`; profiles must support historical resolution.
- **Threats 11–13** (§6). Selective recording, input-channel manipulation, redaction abuse.

D.3 Clarifications and corrections

- **Canonical artifact encoding defined** (§2.5.1). `jcs+json` and `octet-stream` registered; the NaN/Infinity hazard for JSON outputs stated.
- **Claim-type identifier form** (§2.2.4). Compact `family/name` identifiers legitimized alongside absolute URIs, resolving the v0.6.2 schema/prose contradiction.
- **Prespecification limitation statement** (§2.2.1, §5.1). Observe timestamps record ingestion, not observability; the lock comparison defends against backdated attestations only; profiles must bind exposure

to external events.
- R2 decay (§2.2.3). The time-limited nature of R2 claims stated normatively, with the archival posture as the durable alternative.
- Expanded tool-call representation (§2.2.3). SHOULD-level for regulated profiles, with the input-channel threat as stated motivation.
- Non-goals (§1.3). ″Completeness of unprespecified work″ added; retention non-goal clarified against the bundle's packaging role.

D.4 Deferred and rejected

See Appendix E.3 for the disposition of review feedback not adopted in this version, with reasons.

Appendix E. Versioning Rationale (Informative)

E.1 Why v0.7.0 and not v0.6.3

The decisive fact is that several of the changes this revision needs alter what bytes get hashed: the identity rule, the digest and signature object encodings, and the REQUIRED `output_encoding` field all change the canonical encoding preimage of every step. Under the versioning convention this document series established for itself — v0.6.2's own Appendix D defines the 0.6.x line as ″backward-compatible additions and clarifications; no producer compliant with v0.6.1 is rendered non-compliant″ — changes that re-derive every step identity cannot be a 0.6.x release without making the convention meaningless. A version number is a compatibility promise; breaking the promise quietly inside a patch number would cost the document credibility with exactly the standards-literate reviewers it is circulated to.

A genuine v0.6.3 was considered and rejected. Its admissible contents would have been the errata-class items only: the claim-type URI/compact-name contradiction, the skew-tolerance relaxation, the prespecification limitation statement, the R2-decay language. Shipping those as v0.6.3 and the breaking changes as v0.7.0 weeks later would have (a) asked external reviewers to read the same document twice in one cycle, (b) produced a v0.6.3 whose proofs are obsolete on arrival, and (c) delivered zero benefit to any deployed system, because there are none. With no deployments and no implementations, a compatible release protects nobody; it only doubles review churn.

The deeper principle: **the cost of a breaking change is at its lifetime minimum right now, and it rises monotonically from here.** Identity-rule and encoding changes are the most expensive class of change a content-addressed protocol can make — they invalidate every stored proof and every implementation simultaneously. The correct strategy pre-1.0 is to batch every identity-affecting change into a single revision, take the break once, and then hold the line. v0.6.2 had already implicitly endorsed this by deferring the structured digest and signature objects ″to v0.7″; this revision honors that deferral and adds the identity-rule change to the same batch precisely so that there is one re-encoding event, not two.

E.2 The versioning rule going forward

- 0.7.x releases are backward-compatible additions and clarifications only, in the sense v0.6.2's Appendix D established for 0.6.x: no v0.7.0-conformant proof is rendered non-conformant, and no step identity changes value.
- A 0.8 would be warranted only by a further identity- or encoding-affecting change; none is currently anticipated, and proposals of that class are presumptively deferred to accumulate.
- 1.0 is gated on, at minimum: publication of the conformance test suite and Core Test Profile (Appendix B); a reference verifier passing it; at least one formally published profile; and at least one end-to-end proof produced and verified by parties other than the editor. The patent covenant's interim language (§0.3) retires at the same gate.

E.3 Disposition of review feedback (adopted, adapted, deferred, rejected)

This revision was prepared against a structured external review of v0.6.2. For the record, and because deciding what *not* to specify is most of the job:

Adopted as proposed. Identity over `to_timestamp`; structured digest/signature objects pulled forward; canonical artifact encoding; skew tolerance; signing-time authorization; claim-type identifier fix; coverage predicate; dual-commitment redaction; archival bundle; verification report schema; independence classes; tool-call/rationale carriers; threats 11–13; prespecification limitation statement; R2-decay language; test-suite priority elevation.

Adapted. *Coverage* was proposed as a new conformance predicate; it is specified instead as a conditional extension of the existing prespecification machinery (inventory in the claim body, `analysis_id` binding, L4A requirement 3 vacuous without an inventory) so that the level architecture and migration story stay intact. *Expanded tool-call representation* was proposed as a base-protocol default flip; it is taken as SHOULD-for-regulated-profiles instead, because the blob default remains right for the protocol's unregulated floor and the profile mechanism exists precisely to ratchet such requirements where they earn their cost.

Deferred, with trigger conditions. *Per-field salted commitments / Merkle selective disclosure*: deferred until a profile demonstrates a verifier population that needs field-level confirmation without access authority; the dual-commitment-plus-authorized-verifier model covers the regulator-shaped verifier that exists today, and the encoding cost of the richer mechanism is not hypothetical. *Multi-replay divergence statistics*: left to profile claim types (§3.3); base machinery would freeze a statistical methodology the field has not settled. *Human-readable rendering specification and reviewer's guide*: accepted as necessary for adoption, rejected as base-spec content; they are companion documents, because rendering churns on UX timescales while the evidence format must not, and §11.10(b)-style copy requirements are met by the bundle plus a rendering layer above the protocol.

Rejected. *Divergence semirings in the base protocol*: remains out of scope (§3.3); composition semantics over divergence is a research program, not a stabilization item. *eCTD packaging in the base protocol*: submission-structure placement is a clinical-profile concern (§7.4) — the base protocol defines the bundle, the profile defines where the bundle goes. *Renaming the protocol or the L4A/L4R levels*: naming concerns are noted but identifier churn pre-1.0 buys confusion, not clarity; the scope discipline of §1.4 is the protocol's actual answer to ″proof of insight″ overclaim worries. *Migration of editorial control to a standards body in this revision*: sequencing, not substance — §0.5 already anticipates an independent editorial body; credible neutrality requires the test suite and a second implementation first, and a premature handoff would transfer a document, not a standard.

End of v0.7.0 working draft.