The Spherepop Development Protocol: A Formal Specification for History-Preserving Software Engineering

Flyxion
Independent Researcher

Specification version 0.1, November 2026

Abstract

Generative systems have lowered the cost of producing candidate software artifacts much faster than the cost of deciding which candidates deserve to remain in a project. This specification defines a protocol that separates the construction of a proposed change from its acceptance into an authoritative history. Every change is enclosed in a development sphere, a region of provisional computation bound to an explicit specification, whose results are committed only when a finite certificate is accepted under registered checking procedures and an authorization is held against the current head. The protocol adapts the six Spherepop operations to development work, states non-entailment results separating generation, test success, agreement, contribution and authority, and specifies evidence binding, typed refusals, nested composition, a maintenance budget, governance amendment and a threat model. A worked example is supplied with a reproducible fixture. The document specifies a prospective protocol. A limited executable reference implementation exercises selected historical and commitment semantics, and its existence does not establish conformance with the complete specification or the effectiveness of the proposed methodology.

Keywords: software engineering, event histories, specification, verification, maintenance, provenance, portability

1 Introduction and Scope

Software projects have always depended on an unstated balance between the effort of proposing a change and the effort of judging it. Contributors who wanted to add a feature paid for the proposal with their own time, and that cost imposed an incidental limit on how many proposals reached the people responsible for the long-term coherence of a project. Generative coding systems remove much of that limit without removing the responsibility. A patch that takes minutes to produce can still impose an obligation of indefinite length on whoever accepts it, and the review of that patch still requires judgment about whether the feature belongs in the project, whether it introduces dependencies, whether it weakens architectural invariants, and who will bear its upkeep.

A recent interview of Chris Lattner on the Digital Disruption program [1] states three structural problems that this specification takes as design requirements. The first is an asymmetry between production and maintenance, in which output increases at one stage of the process while the capacity of the following stage stays fixed. The second is the difference between specifying a familiar artifact by reference to a convention and specifying something novel, for which natural language leaves consequential properties undetermined. The third is the fragmentation of computational infrastructure across accelerators, which duplicates engineering effort and ties applications to particular vendors. The interview is treated here as a source of problem statements and not as quantitative evidence of industry-wide trends.

The protocol responds by moving the central object of software engineering from the most recently generated version of the source files to the admissible history from which those files are derived. A model, a human contributor or a script may propose any number of transformations. None of them acquires authority over the project by being proposed, by running successfully, or by being accompanied by a report that says it was verified. Authority is conferred only by a committed event whose preconditions the kernel has itself checked.

The specification builds on the six-operation formulation of Spherepop given in Chapter 6 of *Distinction Under Constraint* [17]. In that calculus POP creates a disposition in the open state from a well-formed proposal, BIND attaches an authorization and records the base version, TRANSFORM computes a tentative successor held privately in the disposition's sphere, VERIFY applies a deterministic checker to evidence, COLLAPSE is the only operation that assigns to the committed state, and REFUSE moves a live disposition to the refused state with a reason. It also builds on the history-based treatment of programs in *Programs as Histories* [18], including the strict growth of history under execution and the typing of terms by admissibility. The term replay equivalence is used here for the deterministic replay of an event history, which is related to but distinct from the equality of compiler and interpreter histories in that work. The development-specific types, predicates, budgets and interfaces introduced below are new design proposals layered on that calculus. They are not claimed to be consequences of it, and no part of this document asserts that a complete conforming implementation exists.

The concerns are continuous with an earlier argument about the degradation of software under unreviewed generation [19], and this document is its constructive counterpart. A

specification fixes some properties of an implementation and leaves others open, so producing an implementation settles open properties without anyone deciding them, and a protocol for software engineering must treat that gap as a first-class object.

Scope. The protocol is independent of programming language, editor, model provider and hardware architecture. It governs how changes are proposed, evidenced, authorized, committed, repaired and observed. It does not prescribe how a verifier should check any particular property, how a model should generate code, or how a community should constitute its governance, although it requires that each of these be explicit and recorded.

The protocol applies to every kind of proposer. A human programmer, a conventional compiler, a shell script, a theorem prover and a language model meet the same boundaries when they propose a transformation, and their internal mechanisms are irrelevant to the conditions governing the effect of the proposal on the authoritative project. Systems that regulate computation through an optimized reward signal shape behavior by adjusting incentives. This protocol instead regulates it by explicit permissions, constraints, evidence and admissible continuations, so that no mechanism needs to be taught to prefer an open boundary to a closed one.

What is distinctive. Event sourcing, content addressing, capability-based authorization, continuous integration and signed commits each have established precedents, and a reader could take the protocol to be their combination. The contribution claimed here is narrower and lies in the semantics of provisional computational regions. A sphere is not a branch. It is a region in which computation proceeds without acquiring authority over its enclosing state, and in which nested regions are composed only when boundary obligations, some of them invisible within any single subordinate region, have been discharged. The protocol supplies a compositional language for distinguishing local computation, local certification, global admissibility and authoritative continuation (Sections 6 and 11).

Status. This document specifies a prospective development protocol. A limited executable reference implementation, consisting of two independently written kernels and three fixtures, exercises selected historical and commitment semantics (Section 18). Its existence does not establish conformance with the complete specification and does not demonstrate the effectiveness of the proposed development methodology.

Established work. The architecture uses mechanisms with precedents. Proof-carrying code separates an untrusted producer from a trusted checker [8]. Design by contract, rely-guarantee reasoning and assume-guarantee verification address the conditions under which independently specified components compose [9, 10, 11]. Capability security, optimistic concurrency control and event sourcing supply the authorization, commit and history mechanisms [12, 13, 14]. Software supply-chain attestations and reproducible builds bind evidence to artifact identities and declared environments [15, 16]. These are established mechanisms. The synthesis claimed

here is their organization into a semantics of provisional regions, boundary obligations, typed obstruction and authoritative continuation.

Organization. Section 2 states normative conventions and core requirements. Sections 3 to 6 define the project, the sphere, the operations and nested composition. Sections 7 to 10 treat specifications, refusal, evidence and governance. Section 11 collects mathematical results. Sections 12 to 16 treat maintenance, evaluation, portability, human roles and repair. Sections 17 to 19 give the threat model, the reference architecture and a worked example. Section 20 describes conformance and a comparative study, and Section 21 states limitations.

2 Normative Conventions and Core Requirements

The words MUST, MUST NOT, SHOULD, SHOULD NOT and MAY are used with the meanings given in RFC 2119 [2]. Explanatory text that does not use these words is informative. Three conformance classes are recognized. A *kernel* maintains the history, performs replay and decides whether a proposed commitment satisfies its preconditions. A *verifier* produces evidence about a proposed transformation under a declared procedure. An *adapter* translates between the protocol and an external system such as Git, a compiler, an editor or a model interface. A *reduced profile* for small projects is a fourth conformance class. It retains the history, the three structures, the disposition record, canonical encoding, atomic commitment and the separation of generation from commitment, and it omits budgets, multidimensional evaluation and portability obligations. A system claiming the reduced profile MUST say so, and the requirements on the omitted features do not apply to it. A single program may implement more than one class, but the roles it plays in a given event MUST be distinguishable in the record.

The following requirements apply to every conforming development system. They are stated before any formalism because the formalism exists to make them checkable.

Requirement 1. A conforming system MUST preserve the distinction among proposal, computation, verification, authorization, commitment and observation. Each of these MUST be a separately recorded fact about a change.

Requirement 2. The generation of a software artifact MUST NOT constitute authorization to integrate that artifact into the project.

Requirement 3. Successful execution of an artifact, including successful execution of its tests, MUST NOT be treated as sufficient evidence that it satisfies its specification unless an argument registered with the verification procedure establishes that the executions cover the required property, as exhaustive testing of a finite domain can.

Requirement 4. A verification report MUST NOT be treated as evidence of its own correctness. A report is evidence only relative to a declared checking procedure that the kernel, or a verifier independent of the report's producer, can apply.

Requirement 5. Computational contribution MUST NOT automatically confer authorship, institutional standing or decision-making authority.

Requirement 6. The authoritative state of a project MUST be a function of its admissible history and of nothing else. No projection of that history, however convenient, MAY acquire authority to modify it.

Requirement 7. When a proposed continuation cannot proceed, the system MUST record a refusal that states the reason and carries a checkable certificate, and MUST NOT discard the attempt silently.

Requirement 8. Evidence MUST be rejected as stale when the history it was computed against has advanced in a way that can affect it.

Requirement 9. Ambiguities in a specification that affect consequential properties MUST be preserved as explicit objects until an authorized refinement resolves them.

Requirement 10. The system MUST NOT resolve incomparable evaluation outcomes through an undisclosed weighting.

Requirement 11. Every commitment MUST be checked against the authoritative head at the moment it is appended, and the check and the append MUST form one atomic transaction.

Requirement 12. Authorization of a change to the authorization environment, to the invariant registry or to a specification MUST be evaluated against the state before the change and never against the state the change would produce.

Requirement 13. The kernel MUST decide only certification and authorization. Failure to establish admissibility MUST NOT be treated as establishing inadmissibility, and the receipt MUST preserve which of the two occurred.

Requirement 14. Events MUST have a canonical encoding, so that two serializations of the same event cannot yield different identities.

3 The Project as a Historical Object

Definition 1 (Event and history)**.** *An* event *is a tuple* $e = (k, \pi, d, \varepsilon, \alpha)$ *in which* k *is a kind drawn from a fixed registry,* π *is a payload,* d *is the digest of the preceding event,* ε *is a set of references to evidence records, and* α *is a reference to an authorization record. A* history *is a finite sequence* $H = (e_1, \ldots, e_n)$ *whose digests form a chain*

$$d_0 = \text{root}, \qquad d_i = \text{Hash}\big(d_{i-1}, \text{enc}(e_i)\big),$$

where each e_i *carries* d_{i-1} *as its parent digest,* Hash *is a collision-resistant function fixed by the protocol profile, and* enc *is a canonical encoding fixed by the same profile, so that equivalent events cannot receive different digests through differences of serialization.*

Let S denote the set of project states and s_0 a distinguished initial state. A transition function $\delta : S \times E \rightharpoonup S$ is partial, because an event can be inapplicable in a state. The replay of a

history is

$$\mathrm{Replay}(H) = \delta(\cdots \delta(\delta(s_0, e_1), e_2) \cdots, e_n),$$

which is defined if and only if every step is defined. The function δ MUST be pure. It MUST NOT read a clock, a random source, the network or any other ambient input. A computation that depends on such an input MUST record the value it observed in the payload of the event that uses it, so that replay does not depend on the world.

Definition 2 (Project)**.** *A project is a tuple*

$$\mathcal{P} = (H, \Sigma, \Gamma, \mathcal{I}, \mathcal{B}, \mathcal{G}),$$

where H is the authoritative event history, Σ is the specification registry, Γ is the authorization environment, $\mathcal{I}$ is the invariant registry, $\mathcal{B}$ is the resource and maintenance budget, and $\mathcal{G}$ is the governance structure.

The five registries are not independent inputs. Each is itself a function of H, so that a change to what is authorized, which invariants hold, how large the budget is, or who may change governance is a committed event subject to the same preconditions as a change to source code. The tuple is written out only because the registries play distinct roles in the admissibility predicate.

Definition 3 (Three historical structures)**.** *A project maintains three append-only structures. The* authoritative history *H is the sequence of committed events described above. For each sphere s, the* provisional journal *J_s records the computation performed inside the sphere and carries no authority. The* disposition and audit ledger *L records receipts, refusals and commitment records. Evidence is bound to the digest of H and never to the digest of L or of any journal.*

The separation resolves what would otherwise be a contradiction in the operational semantics. TRANSFORM, VERIFY and provisional BIND extend J_s. REFUSE extends L. COLLAPSE atomically extends H with a transaction that cites the relevant journal and evidence, and appends the corresponding commitment record to L. An amendment to the specification registry is a BIND whose effect is provisional in J_s and becomes authoritative only through COLLAPSE. A consequence is that rejected proposals never alter the digest of H and therefore never invalidate evidence that would otherwise remain applicable.

The complete replayed state and the observable source tree are derived rather than stored:

$$S_H = \mathrm{Replay}(H), \qquad X_H = \Pi_{\mathrm{source}}(S_H).$$

The state S_H also reconstructs the registries, obligations and receipts from events, so two projects can have identical source trees X_H and different authorization histories. Source files, documentation, binaries, dependency graphs, dashboards and architectural descriptions are projections. For a projection operator Π_k the corresponding view is

$$V_k = \Pi_k(H).$$

A projection MUST NOT acquire authority over H merely because it represents H. An editor buffer, a Git working tree and a rendered dashboard are all in this position. They may be used to construct proposals, and a proposal derived from one is judged exactly as any other.

Histories are ordered by extension. Write $H \preceq H'$ when H is a prefix of H'. An authoritative history MUST change only by extension. No operation of the protocol removes, reorders or rewrites an event of H, and the repair mechanism of Section 16 is accordingly defined as the appending of new events. Collision resistance gives computational assurance and not mathematical injectivity, and holding a digest is not holding the history behind it. The claim used throughout is therefore the following. Two parties that possess complete, valid histories with the same terminal digest, under a common genesis event, canonical encoding, protocol profile and collision-resistant hash, have computational assurance that the histories are identical, and, because δ is pure, that they compute the same state. This is the sense in which replay equivalence is used.

Purity of δ is necessary for independent replay and not sufficient. A history references objects such as historical verifier implementations, specification versions, dependency artifacts, environment descriptions and encodings, and a digest establishes the identity of an object under the hash assumptions without establishing that it remains retrievable. Three properties are therefore distinguished: Integrity(H), that the chain and every referenced digest are consistent; Availability(H), that every object referenced by H can be retrieved; and Replayable(H), that Replay(H) can be computed. A history can retain integrity while ceasing to be replayable because an externally referenced artifact has disappeared. Replaying the historical decisions, which recomputes states from events and recorded evidence references, is inexpensive and deterministic. Rerunning the computations that originally justified those decisions may need historical environments or hardware that no longer exists. Reconstruction of a source tree MUST NOT require the rerunning of historical verification, and the protocol treats recorded evidence as the justification of record.

4 The Development Sphere

Every proposed change is enclosed in a region of provisional computation called a development sphere.

Definition 4 (Development sphere)**.** *A development sphere is a tuple*

$$s = (i, h, p, \sigma, \Delta, c, e, a),$$

where i identifies the sphere, h identifies the parent history digest, p records its purpose, σ identifies the governing specification, Δ describes the proposed transformation, c specifies constraints, e contains evidence gathered so far, and a identifies the authorization requirements for committing it.

A sphere is neither a task in a tracker nor a bundle of files. It is a bounded region within which

alternative implementations may be constructed, compared, rejected or revised without any of that activity touching H. A sphere MAY contain subordinate spheres for individual modules, proofs, tests, platform realizations or documentation changes, and the subordinate spheres inherit the constraints of the enclosing one. Provisional events inside a sphere are kept in a sphere journal that is itself content addressed, so that the work remains inspectable even when the sphere is later refused.

A sphere carries a set of *boundary obligations*, which are the predicates that must hold before the sphere can dissolve into committed history. They include the existence of a bound specification, the presence of the evidence the specification requires, the satisfaction of the invariant registry, the registration of any maintenance obligations created, and the presence of a valid authorization. Dissolution into history is permitted only when every boundary obligation has been discharged.

When the authoritative head advances, an open sphere whose base digest is no longer the head is not invalidated, and it is not silently carried forward. It is *rebased*. Rebasing a sphere s with base h onto a head $h' \succeq h$ is an operation that creates a new candidate τ' derived from τ against h', preserves the old candidate and its evidence in the journal, and treats every evidence record whose applicability fails against (h', τ') as unavailable. The rebased candidate has its own digest and requires evidence of its own, as the worked example of Section 19 shows. A sphere that cannot be rebased, because the events since h make its specification binding or its transformation inapplicable, receives a stale disposition. Rebasing is a TRANSFORM and confers no authority.

The number of simultaneously open spheres is bounded by policy. Let K be the maximum number of open spheres a governance structure allows per reviewing principal. A proposal that would exceed that bound is not admitted to a journal at all. It receives a deferred disposition whose receipt may be aggregated with others of the same kind (Section 17), and the disposition says nothing about the merits of the proposal. This is the protocol's response to the queueing problem in which contributions arrive at rate λ and are evaluated at rate μ, so that an unreviewed backlog B grows as $dB/dt = \lambda - \mu$ whenever the queue is nonempty. The protocol cannot raise μ. It can keep the backlog outside the authoritative history, bound its size, and make each rejection cheap and informative.

5 Operational Semantics

The six Spherepop operations receive the following development-specific interpretations. These are specializations of the general calculus in [17] and not replacements for its definitions.

Operation	Development interpretation	Structure extended
POP	Open a sphere from a well-formed proposal, creating a disposition in the open state.	Creates J_s; appends a record to L.
REFUSE	Record why a proposed continuation cannot proceed.	Extends L only; H is unchanged.
BIND	Attach the specification, authorization, base version, dependencies, capabilities, invariants and obligations.	Extends J_s; authoritative only through COLLAPSE.
TRANSFORM	Construct or revise a provisional implementation.	Extends J_s.
VERIFY	Check evidence against explicit acceptance conditions.	Extends J_s with an evidence record.
COLLAPSE	Commit a certified and authorized change, the only operation that assigns to the authoritative state.	Extends H atomically for a top-level sphere, or the parent journal J_p for a subordinate one; appends a record to L.

A typical lifecycle is

$$\text{POP} \longrightarrow \text{BIND} \longrightarrow \text{TRANSFORM} \longrightarrow \text{VERIFY} \longrightarrow \begin{cases} \text{COLLAPSE}, \\ \text{REFUSE}. \end{cases}$$

Repeated transformations and verification attempts MAY occur before a terminal disposition, and a failed verification MAY be followed by a new transformation within the same sphere. Projections of the history are not operations of the calculus. A projection has the status of a report and MUST NOT be used as a means of committing a change. In the source calculus a disposition passes through the states open, bound, verified or failed verification, and refused or committed, and COLLAPSE requires a verified disposition and a current base version. The commitment rule below is that requirement in the setting of certificates and authorizations, and the check of the base version is the check against the authoritative head.

The central practical consequence is that an automated coding system may generate thousands of candidate transformations without any of them acquiring authority. COLLAPSE is indexed by its destination. For a top-level sphere, $\text{COLLAPSE}_{s\to H} : H \longrightarrow H'$ creates an authoritative continuation and is available only under the commitment rule below. For a subordinate sphere s with enclosing sphere p, $\text{COLLAPSE}_{s\to p} : J_p \longrightarrow J_p'$ records an accepted provisional contribution within the parent and does not extend H. Only $\text{COLLAPSE}_{s\to H}$ extends the authoritative history.

Three predicates are kept independent, and the independence is one of the foundations of the protocol.

Definition 5 (Validity, certification and authorization)**.** *Let τ be a transformation constructed within a sphere bound to specification σ.* Validity*,* $\text{Valid}(H, \tau)$*, is actual satisfaction of the substantive requirements: τ is bound to σ in Σ_H; the state* $\text{Replay}(H \,\|\, \tau)$ *is defined and satisfies*

every invariant in $\mathcal{I}_H$; the resource constraints, including the budget rule of Section 12, hold; and every consequential property of τ is covered by a resolved requirement of σ or has been resolved by an authorized refinement. Certification*,* Certified(H, τ, W)*, is acceptance of a finite body of evidence under registered checking procedures. Let Λ be the set of procedures registered in the profile and let W be a finite set of evidence records in the sense of Section 9. The predicate holds if and only if W contains every item of evidence required by σ, each item is applicable to (H, τ), and each item is accepted by the procedure in Λ that it names.* Authorization*,* Authorized(H, τ, α)*, is possession in Γ_H, by the proposing and authorizing principals, of the capabilities that τ exercises and of the approvals that governance requires for it. A profile is* sound *when*

$$\text{Certified}(H, \tau, W) \implies \text{Valid}(H, \tau)$$

for all H, τ and W.

Validity concerns the actual requirements and is not decidable for arbitrary software, so a kernel cannot decide it. A kernel can decide certification and authorization. The three predicates are distinct judgments with different epistemic and institutional meanings. They are not logically independent, since soundness of the profile establishes an implication from certification to validity, but none of them is determined by the others in general. A transformation can be valid without being certified, can be certified under a defective profile without being valid, and can be valid and certified without being authorized. The converse of soundness need not hold, since a correct transformation may lack a certificate that the system can recognize. This yields the principle of conservative commitment: the inability to establish validity is not the same as establishing invalidity. Commitment is governed by the inference rule

$$\frac{\text{Certified}(H, \tau, W) \qquad \text{Authorized}(H, \tau, \alpha)}{H \xrightarrow{\text{COLLAPSE}(\tau, W, \alpha)} H \parallel \tau}$$

in which H is the authoritative head at the moment of commitment. The validity of the conclusion depends on the soundness of the profile and on nothing the kernel can verify about the procedures themselves, so profiles are among the objects that the threat model treats as trusted. A kernel can be conforming while its profile is unsound: it then faithfully applies every registered procedure while those procedures fail to establish the properties they purport to certify. Procedural conformance and semantic soundness are different properties, and the protocol claims only the first of them for a kernel.

Conservative commitment has a cost. A correct transformation can be uncommittable because no registered procedure can certify it, so the expressiveness of the profile determines which work is possible. A weak profile blocks legitimate changes, and a profile that accepts too much commits defective ones. The profile is therefore not neutral infrastructure but a political object, since whoever writes and amends it decides what can be committed. Profiles MUST be registered, versioned and amended under the rules of Section 10, and a transformation refused because no registered procedure applies SHOULD receive the unestablished disposition, so that a pattern of such refusals is visible as evidence that the profile needs extension. The same concern applies to distributed settings. A compare-and-swap on a head digest suffices for a

single kernel, whereas a federated project with several kernels needs an agreed ordering of commitments, which this version does not specify (Section 21).

Two spheres constructed against the same head can each satisfy the premises and still not compose, as Proposition 8 shows. The premises are therefore evaluated against the head at the instant of commitment and not against the head at which the sphere was opened. Checking and appending MUST form one atomic transaction, or use an equivalent compare-and-swap on the head digest. A commitment record identifies both the parent digest that the sphere expected and the parent digest actually found, and a commitment whose two parents differ is accepted only if the evidence applicability rules of Section 9 hold for the actual parent.

6 Nested Spheres and Composition

A sphere s may contain subordinate spheres $s_1, \ldots, s_n$, each bound to its own specification and constructed against its own view of the parent. Each subordinate sphere has a local certificate W_j and a local admissibility judgment relative to the state of the parent at the time of construction. The parent has boundary obligations $\mathrm{ob}(s)$ that are not visible from inside any single subordinate region, because they concern relations among the regions: that interfaces agree, that invariants quantified over the whole are preserved, that the combined maintenance burden remains within budget, and that no two regions have resolved the same open decision incompatibly.

Four notions must therefore be kept apart. *Local computation* is whatever occurs inside a subordinate sphere. *Local certification* is the acceptance of W_j for s_j alone. *Global admissibility* is the satisfaction of the parent's boundary obligations by the composition of the subordinate results. *Authoritative continuation* is the commitment of the composed result to H under the rule of Section 5. Local certification of every subordinate sphere does not entail global admissibility of the composition (Proposition 9), and global admissibility does not entail authoritative continuation without authorization and an unchanged head.

A parent sphere dissolves only through a composition step that records the composed transformation, the certificates of all subordinate spheres, and a certificate for each boundary obligation of the parent. A subordinate sphere is committed locally by $\textsc{collapse}_{s_j \to s}$, which dissolves it into the enclosing provisional sphere when its own boundary obligations are discharged. A locally accepted implementation therefore remains globally provisional, and a parent can refuse a composition without invalidating the subordinate certificates, which were correctly established within their respective scopes. Acceptance is relative to a boundary, and authority does not propagate automatically across enclosing boundaries.

Let $W_{\partial s}$ be a certificate for the relational obligations of the enclosing sphere. The composition predicate $\mathrm{Compatible}(s_1, \ldots, s_n; W_{\partial s})$ holds when $W_{\partial s}$ is accepted for $\mathrm{ob}(s)$ against the subordinate results. The certificate of the parent then has a recursive structure,

$$W_s = (W_1, \ldots, W_n, W_{\partial s}),$$

where each W_j may itself have this form. Associativity of composition is not assumed. That the boundary obligations of a composed result are determined by those of its parts does not alone establish it, since the composition operation and its induced obligations must also be invariant under regrouping. A profile that claims associativity for a class of obligations MUST register the argument for that class.

7 Specification Before Implementation

Every nontrivial transformation MUST be bound to an identifiable specification. A specification is a structured collection

$$\sigma = (F, N, I, E, U),$$

in which F denotes functional requirements, N nonfunctional requirements, I invariants, E the evidence required for acceptance, and U the set of decisions that are explicitly unresolved.

The set U carries the weight of the design. Natural language requests for a conventional artifact succeed largely because the convention supplies the missing detail, whereas a novel system gets no such help, and its consequential properties must be settled by someone. Let $\mathcal{I}(p)$ denote the set of implementations compatible with a prompt or requirement text p, and let Q denote a property of an implementation. The text fails to resolve Q when there are $i_1, i_2 \in \mathcal{I}(p)$ with $Q(i_1) \neq Q(i_2)$. Two applications that satisfy the same request can differ in whether they preserve user privacy, whether they require a network connection, how much memory they consume, and whether they remain usable after a commercial service is withdrawn. Producing one implementation does not resolve these questions. It selects one point in the space of compatible implementations, and a low cost of implementation can conceal the selection by making the result look finished.

A conforming system therefore MUST maintain a *decision manifest* for each transformation, which lists the dependencies introduced, the external services contacted, the data persisted and where, the resource bounds relied upon, and any other property that the governing specification has designated as consequential. The manifest is produced by the proposer and independently recomputed, where the property is statically decidable, by a verifier. Differences between the manifest and the resolved requirements of σ are reported as unresolved decisions.

If two implementations i_1 and i_2 satisfy the resolved requirements of σ but differ on a consequential unresolved property, the system MUST NOT silently represent either of them as uniquely required. An authorized refinement $\sigma \to \sigma'$ that moves the item from U into F, N or I is needed to distinguish them, and further code generation cannot supply it. A refinement is itself a BIND event and is subject to authorization like any other.

8 Refusal and Typed Dispositions

The protocol does not attribute psychological states to computational agents. It regulates their behavior by establishing conditions under which a proposed continuation can be committed. Define the admissible continuation set of a history as

$$\begin{aligned}\mathcal{A}_{\text{valid}}(H) &= \{\tau \in \mathcal{T}(H) : \text{Valid}(H, \tau)\},\\ \mathcal{A}_{\text{authorized}}(H) &= \{\tau \in \mathcal{T}(H) : \exists \alpha\ \text{Authorized}(H, \tau, \alpha)\},\\ \mathcal{A}_{\text{commit}}(H) &= \{\tau \in \mathcal{T}(H) : \exists W, \alpha\ \big(\text{Certified}(H, \tau, W) \wedge \text{Authorized}(H, \tau, \alpha)\big)\}.\end{aligned}$$

where $\mathcal{T}(H)$ is the set of transformations constructible from H. Under a sound profile $\mathcal{A}_{\text{commit}}(H) \subseteq \mathcal{A}_{\text{valid}}(H) \cap \mathcal{A}_{\text{authorized}}(H)$. The inclusion need not be an equality, because a valid and authorized transformation may lack an available certificate, and under an unsound profile even the inclusion is not guaranteed. This matters because the protocol permits a procedurally conforming kernel to operate under a semantically defective profile, as the escaped defect of the worked example illustrates. The kernel decides membership in $\mathcal{A}_{\text{commit}}(H)$, and $\mathcal{A}(H)$ denotes that set below. A transformation outside $\mathcal{A}(H)$ is not punished. It is unavailable for commitment under the present conditions. Verification likewise does not reward an agent. It establishes evidence relevant to a transition. A boundary that stays closed is not a penalty and a boundary that opens is not a prize. The distinction is between altering the relative desirability of an operation and altering its availability. An optimizer that is discouraged from an unauthorized operation by a finite negative reward $r(\tau_{\text{unauth}}) = -C$ may still select it when another component of its objective outweighs C. The protocol instead places the operation outside the commit-eligible set $\mathcal{A}_{\text{commit}}(H)$ defined below. This does not mean that learning systems cannot incorporate hard constraints. It means that the authority boundary is implemented as a constraint outside the proposer and is not entrusted to the proposer's optimization behavior, and the same boundary applies to humans, scripts, compilers, theorem provers and language models without any theory of their motivation. This vocabulary matters because language borrowed from training and discipline invites designers to treat the committing step as an incentive to be tuned, which is the wrong model of what a gate is.

A failed or impossible commitment produces a documented obstruction

$$\textsc{refuse}(\tau, r, w),$$

where r is a typed reason, a pair of a disposition type and a registered cause, and w is a certificate supporting it. The reasons for which a continuation cannot proceed are not epistemically alike. An invariant violation is not the same kind of fact as insufficient evidence, a missing authorization says nothing about the quality of an implementation, and exhausted reviewing capacity says nothing about a proposal's merits. The receipt therefore carries a set of independently supported findings, each of one of the following disposition types, together with an optional primary operational disposition naming the reason that processing stopped. A proposal can be unauthorized, rest on stale evidence and contain a demonstrated invariant violation at once, and a single label would replace a multidimensional judgment by another single

classification. The set also distinguishes properties that were not evaluated from properties that were evaluated negatively, so a proposal stopped at the authorization gate acquires no implicit record of technical incorrectness.

Disposition	Meaning
`ACCEPTABLE`	Required conditions have been established.
`CONTRADICTED`	A required condition has been shown to fail.
`UNESTABLISHED`	Available evidence does not establish a required condition.
`UNAUTHORIZED`	A necessary capability or approval is absent.
`STALE`	Previously applicable evidence is no longer sufficient.
`DEFERRED`	Evaluation cannot presently be completed.
`UNRESOLVED`	A consequential specification decision remains open.

REFUSE remains the general operation that prevents commitment, and its receipt preserves the particular epistemic or institutional condition responsible. A proposal deferred for lack of reviewing capacity thereby acquires no record of technical defect. These types are dispositions of proposed transitions and are not evaluations of the worth of the proposing agent. The certificate MUST be checkable without trusting the party that proposed τ. For an invariant violation it is an input or trace on which the invariant fails. For stale evidence it is the intersecting event. For a missing authorization it is the capability that was required and the principal that lacked it.

A previously refused transformation may later become admissible. A dependency may become available, a specification may be refined, a principal may acquire authorization, or an obligation may be discharged. A receipt is therefore indexed by the context of its refusal as well as by the transformation:

$$\kappa = \mathrm{Hash}\big(\tau,\ d_H,\ \lambda,\ \Lambda_\sigma,\ \sigma,\ \Gamma,\ \mathcal{I},\ \mathcal{B},\ p\big),$$

where d_H is the digest of the head against which the decision was made, λ is the profile version, Λ_σ are the digests of the checking procedures the specification requires, and p is the proposing principal wherever the disposition depends on it, or by a digest of the relevant portions of those structures. Identical transformations evaluated under identical registry values can have different dispositions when the underlying source state differs, which is why the head is included. A later proposal whose κ matches receives the earlier receipt at once, as historical evidence. A proposal that matches τ but not the context receives the earlier receipt marked as relating to a different context, and the earlier receipt never acts as a permanent prohibition. In a sufficiently expressive computational model, deciding whether two programs are semantically equivalent is undecidable by the theorem of Rice [5], although it is decidable for restricted languages and finite-state systems, so in general a fingerprint can only be computed from a syntactic normal form or from a declared abstraction, and the receipt for a near match MUST be presented as related and not as identical. Because two distinct transformations can share a normal form, the fingerprint is required to be conservative. A profile MUST either compute κ from the exact digest of τ, or register a normal form together with an argument that transformations sharing it have the same admissibility under every context in which κ matches. A profile that cannot supply the argument uses the exact digest, and a collision under a registered normal form is a

defect of the profile.

9 Evidence and Certificates

Evidence must be bound to the exact candidate and not merely to a base history. Suppose a generative system produces a patch, runs the tests, and then modifies the patch. The authoritative history has not changed, so evidence tied only to a base digest and a footprint would remain fresh while describing a different artifact. An evidence record is therefore the tuple

$$\omega = (h, t, \sigma, v, \eta, F, r),$$

where h is the digest of the base history, t is the digest of the candidate transformation, σ is the specification version, v is the digest of the verifier implementation, η describes the execution environment, F is the dependency footprint, and r is the result together with the certificate that supports it.

Definition 6 (Applicability)**.** *An evidence record ω is* applicable *to a history H and a transformation τ when t equals the digest of τ; σ is the specification version bound to the sphere; v is registered in the profile and η is among its permitted environments; h is a prefix of H; and no event appended to H after h touches any element of F.*

A footprint F must be a conservative over-approximation of the semantic dependencies of the verification, and not merely a record of what the verifier observed. A test that never exercised a particular configuration may nevertheless depend on it, so an observed footprint is a lower bound and cannot support freshness by itself. When an over-approximation cannot be justified under the verifier's registered procedure, the footprint is taken to be the whole project, and the evidence is applicable only against the exact digest h. An evidence record that omits its footprint is treated in the same way.

Three footprints are distinguished, with $F_{\text{observed}} \subseteq F_{\text{semantic}} \subseteq F_{\text{certified}}$, and only the last can justify reuse of evidence. The certified footprint must also include the dependencies of the verification environment itself: specification versions, verifier registrations, invariant policies, toolchains and external artifacts. A governance amendment that changes what counts as acceptable evidence can invalidate a certificate although no source file has changed. Historical acceptance is distinguished from present applicability. A certificate may have been correctly accepted under the profile in force at the time and be unacceptable under the current one, which does not make the historical record false. The distinction lets a project reconstruct what was justified by the standards available at an earlier time without pretending that those standards remain adequate.

A certificate W is a finite set of evidence records. It is accepted by a registered procedure in Λ when the procedure, applied to the records it names, returns acceptance, and the profile may additionally require declared independence among the verifiers that produced the records. A record is itself only a claim until a registered procedure has applied to it, in accordance with

the fourth core requirement. A finite certificate does not imply that its checking procedure terminates. Every registered procedure MUST declare a checking budget, and exhaustion of the budget or a timeout yields an unestablished or deferred finding and never a finding that the property fails.

The ledger distinguishes receipt metadata from retained certificate payloads. Metadata, which comprises the disposition types, the digests of the transformation and of the evidence, the context index and the head at which the decision was made, is immutable and append-only. Payloads MAY expire under the project's retention policy, and when they do the ledger retains their digests and a record of the expiry, so that the append-only requirement coexists with bounded storage and the remaining record states exactly what has been removed.

Authorization records that carry a validity interval introduce a dependency on time. Replay must remain deterministic, so the time against which an interval is judged MUST be an authenticated time observation recorded in the event that uses it, or the authorization MUST rely on a nonce mechanism instead of an interval.

10 Governance and Amendment

The governance structure $\mathcal{G}$ and the registries Γ, $\mathcal{I}$, Σ and $\mathcal{B}$ are derived from history, so a transformation could in principle modify the authorization environment in a way that grants its own proposer the permission needed to commit it. The specification prohibits this. For every transformation τ_g that modifies governance, authorization is evaluated against the state before the transition,

$$\mathrm{Authorized}(\Gamma_H, \tau_g),$$

and never against $\mathrm{Authorized}(\Gamma_{H\|\tau_g}, \tau_g)$. The same rule applies to the invariant registry, so that a transformation cannot evade an invariant by deleting it within the same transaction, and to specification registry changes. Every admissibility condition of the transaction is evaluated under the registries of H. The amended registries take effect for the next transaction.

Amendments to governance, to invariants and to specifications follow explicit amendment procedures, registered in $\mathcal{G}$, whose authority derives from the preceding state. An amendment SHOULD be committed as its own transaction and SHOULD NOT be bundled with ordinary transformations, since bundling obscures which authorizations were exercised.

The regress of authorizations needs a termination condition. The genesis event of a project designates the initial authorization environment, invariant registry, profile and trust root. Acceptance of the genesis digest is an act of the participants that occurs outside the protocol, and the protocol cannot bootstrap it. All later authority is traceable to that event, and a participant who doubts the genesis event has no recourse inside the protocol except to refuse to follow the history.

11 Non-Entailment Results

The core requirements of Section 2 become checkable once the relations they separate are given formal statements. Throughout, $\mathcal{T}(H)$ is the set of transformations constructible from H and $\mathcal{A}(H) = \mathcal{A}_{\text{commit}}(H) \subseteq \mathcal{T}(H)$ is the commit-eligible subset.

Proposition 1 (Generation does not entail admissibility)**.** *If the profile is sound and the invariant registry or the authorization environment prohibits some constructible transformation, then $\mathcal{A}(H) \subsetneq \mathcal{T}(H)$, and consequently*

$$\tau \in \mathcal{T}(H) \implies \tau \in \mathcal{A}(H).$$

Proof. Let $\tau_0 \in \mathcal{T}(H)$ be a transformation that violates an invariant of $\mathcal{I}_H$ or exercises a capability absent from Γ_H. If τ_0 violates an invariant it is not valid, so by soundness of the profile no certificate exists for it, and if it exercises an absent capability it is not authorized. In either case $\tau_0 \notin \mathcal{A}(H)$, while $\tau_0 \in \mathcal{T}(H)$ by assumption. Hence the inclusion is proper. □

The proposition is elementary and architecturally decisive. It states that the ability of a system to produce a change and the permission of the project to accept it are different relations. Greater generative capability can enlarge $\mathcal{T}(H)$ and so change which admissible transformations are practically reachable, but it cannot by itself confer authorization or establish that the acceptance conditions hold.

Proposition 2 (Finite penalization does not generally enforce an authorization boundary)**.** *Let $\mathcal{T}$ be a finite set of constructible transformations, $\mathcal{A}_{\text{authorized}} \subsetneq \mathcal{T}$ the authorized subset, $U : \mathcal{T} \to \mathbb{R}$ a utility, and $J_C(\tau) = U(\tau) - C\,\mathbf{1}_{\tau \notin \mathcal{A}_{\text{authorized}}}$ for a finite penalty C. If U is not bounded in advance, then for every finite C there is a utility U for which $\arg\max_{\tau \in \mathcal{T}} J_C(\tau) \not\subseteq \mathcal{A}_{\text{authorized}}$. If $\max U - \min U < C$, then $\arg\max J_C \subseteq \mathcal{A}_{\text{authorized}}$. Constrained maximization of U over $\mathcal{A}_{\text{authorized}}$ excludes every unauthorized transformation whatever U is.*

Proof. Fix C and choose $\tau^* \notin \mathcal{A}_{\text{authorized}}$ and $\tau \in \mathcal{A}_{\text{authorized}}$. Set $U(\tau^*) = U(\tau) + C + 1$, so $J_C(\tau^*) = U(\tau) + 1 > J_C(\tau)$ and τ^* outranks τ. For the second claim, an unauthorized τ^* has $J_C(\tau^*) \leq \max U - C < \min U \leq J_C(\tau)$ for every authorized τ, which exists because the authorized subset is taken to be nonempty. The last statement is the definition of the constrained problem. □

The qualification about bounded utility is essential. A sufficiently large finite penalty can enforce a constraint under suitable boundedness assumptions. The advantage of the protocol is that authorization does not depend on establishing such assumptions about the proposer.

Proposition 3 (Passing tests does not entail satisfying the specification)**.** *Let σ require a property φ over a domain D, and let $\mathcal{T} \subseteq D$ be the finite set of cases exercised by a test suite. Suppose some implementation i_1 satisfies φ, and the implementation class contains a modification i_2 of i_1 that agrees with it on $\mathcal{T}$ and violates φ at some point of $D \setminus \mathcal{T}$. Then both pass the suite identically although only i_1 satisfies φ.*

Proof. By hypothesis i_2 returns the same value as i_1 on every case of $\mathcal{T}$, so both pass every test in the suite, and only i_1 satisfies the specification. The hypothesis holds for ordinary programming languages whenever $D \setminus \mathcal{T}$ is nonempty and φ constrains the value returned at some point outside $\mathcal{T}$, since the value returned at a single point can be changed. □

Proposition 4 (Provenance and authorship are defined by different events)**.** *Let* $\mathrm{Prov}(a)$ *be the set of principals appearing in the provenance record of an artifact* a*, and let* $\mathrm{Auth}(a)$ *be the set of principals to which governance has assigned an accountable role for* a*. Provenance is recorded by the kernel from the history, and role assignments are established only by governance events. Then* $\mathrm{Prov}(a) \subseteq \mathrm{Auth}(a)$ *is not guaranteed, and in a conforming system the two sets are defined by different events.*

Proof. By construction, the kernel appends provenance entries whenever a principal proposes or modifies an artifact, and appends role assignments only upon a governance event authorized under $\mathcal{G}$. A principal that proposed a transformation, and appears in $\mathrm{Prov}(a)$, belongs to $\mathrm{Auth}(a)$ only if such a governance event exists. Absence of the event is consistent with every other requirement of the protocol.

The separation is a design decision and the proposition records its consequence. A concrete history shows that the inclusion fails. Take the history of Section 19, in which `proposer-M` appears in the provenance of the candidate τ_2' because it proposed and rebased it. The only role events in that history assign roles to `maintainer-A`. Hence `proposer-M` $\in \mathrm{Prov}(\tau_2')$ while `proposer-M` $\notin \mathrm{Auth}(\tau_2')$. □

Proposition 5 (Agreement does not entail correctness)**.** *Let* i_1 *and* i_2 *be implementations such that* $O(i_1, u) = O(i_2, u)$ *for every input* u *in the specified domain. It does not follow that either satisfies the specification.*

Proof. Let the specification require an output in a set $\mathrm{Acc}(u)$ and suppose both implementations return the same value outside $\mathrm{Acc}(u)$ for some u, as when they reproduce the same misreading of a requirement. They agree everywhere on the domain and both violate the specification. □

The practical form of the last result is that differential verification requires a declared independence class for each verifier. Two generators trained on overlapping data, two verifiers written by one author, or two compilers sharing a front end are not independent, and agreement among them SHOULD be weighted accordingly. The protocol requires the declaration and does not claim to resolve how independence is to be established.

Proposition 6 (Repair preserves history)**.** *If* $H' = H \parallel \mathrm{Repair}(e, r)$ *for an event* e *of* H*, then* $H \preceq H'$*, the event* e *is a member of* H'*, and the digest chain of* H *is unchanged.*

Proof. Repair is defined as the appending of a new event that refers to e by digest. Appending preserves every earlier event and every earlier digest, since d_i depends only on earlier events and on e_i. □

Proposition 7 (Obligation-reducing transformations satisfy the budget rule)**.** *Under the two-regime budget rule of Section 12, if $M^+(H) \leq B$ and $M^+(H \,\|\, \tau) \leq M^+(H)$, then τ satisfies the rule. If $M^+(H) > B$ and $M^+(H \parallel \tau) < M^+(H)$, then τ also satisfies the rule.*

Proof. In the first regime the rule requires $M^+(H \parallel \tau) \leq B$, and $M^+(H \parallel \tau) \leq M^+(H) \leq B$. In the second regime the rule requires exactly the strict reduction that is assumed. □

The proposition ensures that dependency removal, consolidation and dead-code elimination are never blocked by the budget rule, even when a project is already over budget, which is the formal counterpart of the observation that the most valuable use of a generative system is sometimes the deletion of code.

Proposition 8 (Local admissibility is not compositional)**.** *There exist a history H and transformations τ_1, τ_2 such that* $\mathrm{Valid}(H, \tau_1)$ *and* $\mathrm{Valid}(H, \tau_2)$ *hold while* $\mathrm{Valid}(H \parallel \tau_1, \tau_2)$ *fails.*

Proof. Let the budget be B with remaining headroom $\beta = B - M^+(H) > 0$, and let each of τ_1, τ_2 add obligations whose declared upper bound is 0.6β. Each satisfies the budget rule against H. After τ_1 the headroom is $0.4\beta < 0.6\beta$, so τ_2 violates the rule against $H \parallel \tau_1$. A second example uses an invariant requiring that a function have at least one implementation, with two implementations present: τ_1 removes the first and τ_2 removes the second, and each preserves the invariant against H while the second removal does not preserve it against $H \parallel \tau_1$. □

The proposition is the reason that commitment is evaluated at the head and not at the head at which a sphere was opened, and it connects the protocol to the broader question of how locally valid transitions compose into globally valid ones.

Proposition 9 (Local certification of subordinate spheres does not entail admissibility of the parent)**.** *Let s be a sphere with subordinate spheres $s_1, \ldots, s_n$ and composed transformation $\tau = \tau_1 \circ \cdots \circ \tau_n$. Let* $\mathrm{Certified}_c(s_j)$ *denote certification of s_j relative to its context c_j, the state of the parent that the sphere saw, and let* $\mathrm{Valid}_c(s_j)$ *be the corresponding local validity. Under a sound profile the conjunction* $\bigwedge_{j=1}^{n} \mathrm{Certified}_c(s_j)$ *implies* $\bigwedge_{j=1}^{n} \mathrm{Valid}_c(s_j)$*, and it does not in general imply* $\mathrm{Valid}(H, \tau)$ *for the composed transformation.*

Proof. Let the project have an invariant I stating that the exported names of the project are pairwise distinct, held as a boundary obligation of the parent. Let s_1 add to unit `A` an exported function named `parse`, and let s_2 add to unit `B` an exported function with the same name. Against H, in which no export has that name, each of τ_1 and τ_2 preserves I, and each sphere has a local certificate accepted by a procedure that checks name uniqueness within the visible state of that sphere, so each sphere is certified relative to its context and, the profile being sound, locally valid. The composition $\tau_1 \circ \tau_2$ exports `parse` twice and violates I. The obligation is not visible from inside either sphere, since each sees a state in which the name is free, and it can be discharged only at the parent boundary. □

Proposition 10 (Certified admissibility is conservative)**.** *If a profile is sound, then* $\text{Certified}(H, \tau, W)$ *implies* $\text{Valid}(H, \tau)$*, and failure of* $\text{Certified}(H, \tau, W)$ *for a given* W *does not imply* $\neg\,\text{Valid}(H, \tau)$*.*

Proof. The first claim is the definition of soundness. For the second, let τ satisfy every requirement and let W be empty. The procedures in Λ cannot accept an empty certificate for a specification that requires evidence, so $\text{Certified}(H, \tau, W)$ fails although $\text{Valid}(H, \tau)$ holds. □

Proposition 11 (No circular authorization)**.** *Let* τ_g *be a governance transformation with* $\neg\,\text{Authorized}(H, \tau_g, \alpha)$ *and* $\text{Authorized}(H \parallel \tau_g, \tau_g, \alpha)$*, so that* τ_g *would create the capability needed to commit itself. The second judgment does not justify commitment of* τ_g*.*

Proof. The commitment rule requires $\text{Authorized}(H, \tau_g, \alpha)$ at the head H at the moment of commitment. The assumed failure of that premise prevents the rule from applying, so the transaction is not committed and the history is unchanged. □

The proposition preserves ordinary delegation. A principal without the capability to amend Γ may propose such an amendment, and a maintainer who holds the capability in Γ_H may approve it, so that the capability is then created under authority that already existed. What is prohibited is the derivation of authority from the consequences of the action that the authority is supposed to license.

12 Maintenance Obligations as First-Class Objects

Every accepted change creates a potential future obligation, and the protocol requires that the obligation be registered when the change is committed. Define an obligation as

$$o = (x, r, d, k, b),$$

where x is the artifact concerned, r identifies the responsible role, d specifies review or expiry conditions, k is a category, and b records an estimated resource requirement. The categories include dependency upkeep, interface stability, security response, documentation currency, platform support and data migration.

Because future cost cannot be known exactly, b is an interval $[b^-, b^+]$ with a declared basis for the estimate, and the burden of a history is computed from the upper endpoints:

$$M^+(H) = \sum_{o \in O_{\text{active}}(H)} b^+(o).$$

An unconditional constraint $M^+(H \parallel \tau) \leq B$ would prevent a project that is already over budget from repairing itself in steps. Suppose the budget is 100 units and a revised estimate puts the burden at 140. A transformation that reduces the burden to 120 would be refused because $120 > 100$, although it is a partial repair, and a later one reducing it to 90 would restore

compliance. These figures are illustrative. The budget condition is accordingly a two-regime rule:

$$\mathrm{BudgetOK}(H,\tau) = \begin{cases} M^+(H \parallel \tau) \le B, & M^+(H) \le B, \\ M^+(H \parallel \tau) < M^+(H), & M^+(H) > B, \end{cases}$$

where the inequalities apply to declared estimates under the project's budgeting policy. The second regime permits strictly obligation-reducing transformations while the project remains over budget. It is an instance of a general principle: a system that has entered a condition in which ordinary continuations are inadmissible must retain access to corrective continuations, and must not be required to achieve complete restoration in a single atomic transformation. Proposition 7 states the resulting guarantee.

The budget rule governs the creation of obligations and must not become an obstacle to recognizing obligations that already exist. Three categories of change are therefore distinguished. *Obligation-creating transformations* are subject to the budget rule above. *Evidentiary corrections* revise the recorded estimate of an existing obligation, or register an obligation that was previously unaccounted for. *Amendments to budget policy* change B or the estimation procedure. Suppose a project records a burden of 80 units against a budget of 100, and an audit finds that the burden should be 140. The audit has not created 60 units of work. It has disclosed work that was always there. An evidentiary correction is accepted under its own conditions, namely independent review of the revised estimate as required above and a recorded basis, and it MAY raise the recorded burden above B. The project then operates in the second regime of the rule. A constraint on the creation of liabilities must not become a constraint on the truthful recognition of existing ones, since otherwise accurate accounting would be penalized and inaccurate accounting preserved.

A fourth category is needed for liveness. Suppose a project is over budget and requires an urgent security correction that leaves its burden unchanged, or that temporarily increases it while preventing a catastrophic failure. The strict reduction of the second regime would block the correction. *Protective transformations* are therefore admitted under a separate certificate that establishes the protective purpose, the temporary burden introduced, and the conditions under which the added obligation must be reviewed or discharged. This is not an administrative override. It is a narrowly scoped exception that is authorized by a principal other than the proposer, recorded in the history with its review condition, and counted in M^+ from the moment of commitment.

The scalar budget presumes that obligations measured in reviewer hours, money, storage, computation and specialist availability can be added. A profile MUST therefore define a common unit and an estimation horizon and state its conversion policy. A later version may replace the scalar by a vector $\mathbf{M}^+(H) = (M_1^+(H), \ldots, M_k^+(H))$ with componentwise budgets and authorized trade-offs, in line with the rejection of a universal scalar in Section 13. An obligation whose review date d has passed without a review event MUST be flagged, and governance MAY assign overdue obligations a surcharge. Estimates are revised by explicit events, and a revision that raises the burden above the budget triggers a refusal of subsequent additive proposals until the burden is discharged or the budget is raised by an authorized

governance event.

The rule prevents a development system from treating the introduction of additional functionality as free, and it changes who bears the cost of proposing. A contributor's sphere carries the maintenance delta of the change, which makes the externalized cost visible at the time of proposal, when a reviewer can still decline it. Because a proposer benefits from underestimating b^+, estimates attached to a transformation are not self-certifying. The profile MUST require that estimates above a threshold set by governance be reviewed by a principal other than the proposer, and the review is recorded as evidence under the rules of Section 9. Otherwise the budget would be a fiction maintained by the party it constrains. The estimates remain estimates. The protocol offers a place to record and revise them, and it does not claim that the record is accurate.

13 Evaluation Without a Universal Scalar

Counting lines generated, tokens consumed, commits produced or tests passed measures activity, and activity is a poor proxy for value when its cost approaches zero. Let x_t be the project state at step t, let $d(x_{t+1}, x_t)$ measure the magnitude of a modification, and let V be some evaluation. Total activity

$$A_T = \sum_{t=0}^{T-1} d(x_{t+1}, x_t)$$

and progress

$$P_T = V(x_T) - V(x_0)$$

are not related by any general implication, so that large activity does not generally entail positive progress, and a large A_T is compatible with a nonpositive P_T because a trajectory can return to a functionally equivalent state. This is a familiar instance of the pattern that a measure which becomes a target ceases to be a good measure [3]. The protocol therefore records activity in the ledger as a cost and excludes it from the dimensions on which a project state is evaluated.

A system MUST NOT evaluate development solely through scalar quantities. Each candidate project state is instead associated with a vector

$$\mathbf{Q}(H) = (Q_1(H), \ldots, Q_8(H)),$$

whose components are functional adequacy, verification coverage, maintainability, portability, resource efficiency, dependency exposure, recoverability and accessibility. Each component has a declared measurement procedure and a declared partial order on its values. A governance structure MAY add components or refine them, but MUST NOT remove a component without an authorized event.

A transformation τ is a *Pareto improvement* when $Q_k(H \parallel \tau) \succeq_k Q_k(H)$ for every component and the inequality is strict for at least one. A system MAY identify Pareto improvements when

the relevant components are measurable and ordered. When a transformation improves some components and degrades others, the alternatives are incomparable, and the system MUST NOT choose between them through an undisclosed weighting. Resolution of the trade-off is an explicit governance event: a BIND that records the weights or priority order adopted, the principals who authorized it, and the scope in which it applies. A weighting adopted in this way is itself a project artifact that can be reviewed and revoked.

14 Portability Across Realizations

A project may have independently developed realizations in Rust, Python, Haskell, C++, Mojo or WebAssembly. The protocol does not compete with a portable programming language or a portable compiler infrastructure such as LLVM [4], which seek to preserve behavior across hardware. It operates at a different level, where the aim is to preserve the specification, the provenance, the authorization and the admissibility of a project across the environments in which it is realized. A language such as Mojo MAY serve as a backend without becoming the authoritative definition of the project.

Let $C_b(H, \sigma)$ denote a realization of the project for backend b, and let O_b be an agreed observation function mapping a realization and an input u to an observable outcome in a space Ω. The space Ω may consist of output values, execution traces or sets of possible traces, since a single deterministic output cannot characterize concurrent, probabilistic, interactive or stateful software. A first formulation of a portability obligation requires

$$O_b\big(C_b(H, \sigma), u\big) \simeq_\sigma O_{b'}\big(C_{b'}(H, \sigma), u\big)$$

for all u in the specified domain. The relation $\simeq_\sigma$ must absorb numerical tolerances, permitted nondeterminism, platform capabilities and resource differences. Defined by a tolerance, however, it is not transitive, since outcomes within ϵ of each other pairwise can drift apart across a chain of backends, and a pairwise formulation therefore cannot be a basis for a conformance claim over many backends.

The protocol accordingly adopts a formulation based on acceptable sets. The specification declares, for each input u in its domain, a set $\mathrm{Acc}_\sigma(u) \subseteq \Omega$ of acceptable outcomes, which may be defined by tolerance, by a predicate or by a reference computation. A realization *conforms* when

$$O_b\big(C_b(H, \sigma), u\big) \in \mathrm{Acc}_\sigma(u) \qquad \text{for all } u \in \mathrm{Dom}(\sigma).$$

Portability is then the conformance of every declared backend to the same acceptable sets, which does not depend on transitivity. For m backends, pairwise comparison requires $\binom{m}{2} = m(m-1)/2$ relationships per specification family, whereas checking each backend against the common acceptance criterion requires m. Over n specification families these are $O(nm^2)$ and $O(nm)$ respectively. A stronger count of $n + m$ relationships applies only if a shared intermediate abstraction genuinely separates specification-facing from backend-facing integration work. That separation is an architectural hypothesis with conditions of its own, and acceptable-set semantics

does not imply it. These are counts of relationships and not measurements of development cost, and real systems contain hardware-specific behavior that prevents complete separation.

The objective is not to require identical machine instructions. It is to preserve the distinctions that the specification identifies as consequential. The protocol also cannot address a limit of a different kind. A portable ecosystem can weaken dependence on any one hardware vendor and still leave most individuals unable to obtain or operate the hardware, so conformance of backends is no evidence of access to them.

15 Human Collaboration and Accountable Authorship

The protocol distinguishes computational provenance from human contribution and responsibility. Every committed artifact MUST identify its originating proposal, its transformation history, its verification evidence and its authorization record. Appearing in a provenance record MUST NOT automatically entail authorship, as Proposition 4 states.

Human principals may occupy recognized roles, and the initial set contains author, editor, reviewer, maintainer, verifier and specification steward. A principal MAY occupy several roles, and each role carries its own permissions and obligations. Role assignments are relations among principals, roles and scopes that are established by governance events, so that being a maintainer of one module confers nothing over another. Computational principals MAY hold capabilities to propose, to run verification procedures and to construct projections. Governance can assign a role label, but a label does not establish the capacity to discharge the obligations that come with the role. The protocol therefore distinguishes Assigned(p, r) from Qualified(p, r), and an accountable role assignment is valid only when both hold. For a computational principal, operational capabilities such as proposing changes, running tests or producing certificates can be defined and tested, and none of them establishes the institutional conditions of accountable authorship. Every commitment that carries substantive accountability MUST identify a principal that is qualified to assume the corresponding obligations. An organization cannot satisfy this requirement by declaring its coding system to be the responsible maintainer.

The system supports counterproposals, minority reports, unresolved objections, alternative implementations and competing explanatory treatments without prematurely collapsing them into a single canonical artifact. Alternatives live in parallel spheres or branches of the history, and the designation of one as canonical is a BIND event that points to it, records who authorized the designation, and can later be revised without erasing the alternatives. An objection raised by a reviewer is an event with a reference to the artifact it concerns, and it remains attached to the history after the artifact is accepted.

The same structure applies to work that is not software. A textbook chapter, a mathematical proof, an executable implementation and an associated experiment can be bound into one history while their authorship and verification requirements remain differentiated. The proof is checked by a proof checker, the implementation by tests and conformance runs, the experiment by a replication procedure, and the chapter by human reviewers, each with its own evidence

type under the same disposition semantics.

16 Historical Repair

A defective implementation does not require the destruction of its developmental history. If an accepted change is later found to be defective, the system records a repair event

$$H' = H \parallel \text{Repair}(e, r),$$

where e identifies the defective event and r is the repair, which is itself a transformation subject to admissibility. A repair event MUST name an event that exists in H. A repair whose target is absent is refused with an unestablished finding and leaves H unchanged. The defective event remains identifiable even when its effects are superseded in the current observable state, and a defect receipt records how the defect was discovered, what it affected and which evidence would have caught it earlier.

Two histories can therefore have identical current source trees and different evidentiary content. In one, the code has never encountered a given failure. In the other it encountered the failure, documented it, and incorporated a repair. Define source equivalence $H \approx H'$ by $X_H = X_{H'}$ and historical equivalence by identity of digests. Source equivalence is weaker than executable or observational equivalence, since identical source files can behave differently under different dependencies, compiler versions, build configurations or runtime environments, and those are recorded in S_H and not in X_H. The protocol treats $\approx$ as the relation that matters for comparing source trees, and historical equivalence as the relation that matters for deciding what the project knows about itself. The escaped-defect record of Section 20 is computed from repair events.

An escaped defect can reveal that a checking procedure was inadequate for the property it certified, and in that case two repair obligations arise: one concerning the defective software, and one concerning the procedure that permitted its acceptance. A repair of the artifact alone would leave uncorrected the mechanism through which the defect entered the authoritative history. The corrected procedure is recorded by a governance amendment to the profile, so that a project accumulates corrected acceptance procedures as well as corrected implementations.

Git already preserves considerable history [6], but it does not by itself enforce a distinction among specifications, proposals, evidence, obligations and authorization. The protocol makes those distinctions part of the application semantics while leaving Git in place as a store, a transport and a familiar interface.

17 Threat Model

The kernel is described as small and auditable, and that description is only meaningful relative to what it must protect against. A generative system is treated as an unreliable proposer

whose outputs may contain arbitrary code, misleading evidence, unauthorized operations and concealed dependencies. The kernel, the hash function, the registered checking procedures, the protocol profile, the signing keys of principals and the genesis event form the trusted base. Everything else is untrusted, including verifiers, which are trusted only to the extent that a registered procedure and a declared independence class say.

Defective or malicious proposers. A proposer may submit transformations that exploit the verification environment, exfiltrate data, or attempt to modify the kernel or the ledger. Journals MUST be built in isolated execution environments with no write access to H or L, and verifiers MUST run candidate code in environments described by η.

Forged evidence and compromised verifiers. Evidence is bound to the candidate digest, the verifier digest and the base history, so evidence cannot be moved from one candidate to another. A compromised verifier can still produce false acceptances. The profile SHOULD require acceptance by verifiers of declared independence classes for properties whose failure is costly, and it MUST require that verifier implementations be identified by digest and registered through governance.

Replayed authorization. An authorization record MUST bind the principal, the transformation digest, the expected parent digest and a validity interval or nonce, so that an authorization issued for one commitment cannot be replayed for another.

Unauthorized amendments and races. Section 10 prevents self-authorizing amendments, and Section 5 requires atomic commitment against the current head, which removes races between concurrent commitments.

Dependency substitution. The decision manifest of a transformation names dependencies by content digest. A dependency referred to by a mutable name MUST be resolved to a digest at proposal time, and the resolution is part of the evidence.

Divergence between projections and the ledger. A Git repository, working tree or hosted copy is a projection. If a projection differs from the replay of H, the adapter MUST treat the difference as a new proposal and not as a change to H, and it MUST report the divergence.

Equivocation. A hash chain gives integrity relative to a trusted head or checkpoint. It does not prevent an operator from presenting two different, internally consistent histories to different participants. The single-kernel model excludes such equivocation by the trusted kernel. A deployment that cannot exclude it SHOULD require participants to retain externally signed

checkpoints of the head and to exchange them, since two signed checkpoints that name the same position and different digests are evidence of equivocation.

Resource exhaustion and the audit requirement. Retaining every refused attempt would let an untrusted proposer force the retention of unbounded numbers of receipts, so that the auditability requirement becomes a means of denial of service. Bounding the number of open spheres does not bound the rate of refused proposals. Admission control MUST therefore precede the creation of journals and the running of verification. The profile MAY permit rate limits per principal, storage quotas, content-addressed deduplication of identical proposals, and the aggregation of deferred receipts into a single record that states the principal, the window and the count of proposals deferred. Receipts of types that carry a certificate, such as contradicted or stale proposals, are retained individually for as long as the retention policy of the project requires.

Out of scope. Side channels are outside the model because defending against them requires isolation at the level of the operating system and the hardware, which the protocol cannot specify or check from within its own records. Coercion of maintainers and compromise of the hardware on which the kernel runs are outside the model for the same reason, since in both cases the kernel faithfully records actions that were authorized by a principal who should not have been trusted. The protocol records authorized actions faithfully and cannot make them wise.

18 Reference Architecture

The reference implementation is a local-first development environment that operates alongside Git and does not attempt to replace existing version-control infrastructure. A Git commit is treated as a projection of the protocol history, and the disposition receipt for each commitment is referenced from the commit message, so that a repository without the protocol tooling remains readable and a repository with it remains verifiable.

The authoritative semantics reside in the specification directory and not in any implementation. The repository is organized as follows.

Directory	Contents
`spec/`	Normative specification and formal semantics.
`kernel/`	Event algebra, replay and commitment checking.
`schemas/`	Canonical event and evidence formats.
`ledger/`	Historical records and disposition receipts.
`adapters/`	Interfaces to Git, compilers, editors and model providers.
`verifiers/`	Independent verification components.
`governance/`	Capabilities, authorship, roles and obligations.
`examples/`	Worked software and textbook projects.
`conformance/`	Cross-implementation verification suite.
`benchmarks/`	Proposed only: controlled comparative experiments (Section 20), not yet existing.

The reference kernels are a Rust kernel and an independently written executable specification in Python, which are run against each other as a differential check. They implement a limited subset: event replay, disposition checking, evidence applicability, the budget rule, atomic commitment against the head, and the nested exports example. They do not model journals, rebasing, authorization time or the receipt context index. Their existence does not establish conformance with the complete specification. The kernel is small by design, since the guarantees of the protocol are only as strong as the code that decides whether a commitment is permitted, and a small kernel is the part of the system that can reasonably be audited by those who rely on it. Verifiers, adapters and model interfaces sit outside the kernel and are untrusted except through the evidence they produce.

A resolved sphere is recorded as a canonical, machine-readable disposition. The following record is illustrative of the proposed format.

```
{
  "sphere_id": "parser-0042",
  "parent_history": "sha256:...",
  "purpose": "Remove redundant parser dependencies",
  "specification": "spec:parser@1.3",
  "operations": [
    { "type": "BIND", "reference": "invariant:grammar-preservation" },
    { "type": "TRANSFORM", "artifact": "sha256:..." },
    { "type": "VERIFY", "evidence": "sha256:...",
      "verifier": "parser-conformance" }
  ],
  "authorization": {
    "principal": "maintainer-public-key",
    "policy": "merge-policy@2"
  },
  "provenance": {
    "computational_assistance": true,
    "contribution_record": "sha256:..."
  },
```

```
  "maintenance_delta": {
    "dependencies_removed": 2,
    "dependencies_added": 0
  },
  "disposition": "COLLAPSE"
}
```

The digests identify artifacts and do not establish their correctness. Before accepting the record, the kernel must still validate the evidence against its declared procedure, check the authorization against the policy version named, confirm the freshness of the evidence against the parent history, and verify that each operation was applicable in the sequence recorded. A failed verification produces a record of the same shape with operation REFUSE, a typed disposition from Section 8 and the supporting certificate, so that an unsuccessful attempt is a durable, inspectable fact and not an invisible loss. In a full profile the record also carries the expected and actual parent digests and the context index κ.

A profile MUST name a specific hash function and a specific canonical encoding. The reference profile uses SHA-256 and the encoding below. Because the digest of the parent is a fixed-length string, concatenation with the encoded event is unambiguous, and the separator byte is a delimiter and not a source of domain separation. A profile that hashes objects of several kinds, such as events, evidence records and receipts, SHOULD prefix each input with a distinct type tag so that a digest of one kind cannot be presented as a digest of another. The reference profile as used in the example does not do so, and the fixture records only event digests under the chain.

The reference profile adopts the JSON Canonicalization Scheme of RFC 8785 [7], and additionally prohibits duplicate keys and floating-point numbers in events, since sorted keys and the removal of whitespace do not by themselves settle numeric representation, Unicode handling or escaping. The fixture contains only ASCII strings and integers, for which the scheme coincides with sorted keys, no insignificant whitespace and minimal escaping. The chain digest is $d_i = \text{SHA-256}(d_{i-1} \,\|\, \texttt{0x0A} \,\|\, \text{enc}(e_i))$, with d_{i-1} written as 64 lowercase hexadecimal characters and d_0 the literal string `root`.

19 A Worked Example

A small parser project exercises the historical semantics end to end. The example is also the first conformance fixture, and its machine-readable form accompanies this document with real digests computed under the encoding of Section 18. Digests are abbreviated to twelve hexadecimal characters below. Artifact digests are those of placeholder descriptions and stand in for artifact content. The burden figures are illustrative units.

Genesis. Event e_1 establishes the specification `parser@1.0`, the invariant registry containing `grammar-preservation`, the capabilities (a maintainer `maintainer-A` who may commit and

amend, and a computational proposer `proposer-M` who may propose and run verifiers), a maintenance budget of 100 units, and three obligations with upper bounds 15, 15 and 40 for two library dependencies and the parser core, giving $M^+ = 70$. Its digest is $d_1 =$ `c707dd1034eb`.

A refused proposal. The proposer opens a sphere against d_1 and constructs τ_1 (digest `d1fb1a6b3a1f`), which removes both library dependencies and rewrites the tokenizer. The grammar-conformance verifier produces evidence ω_1 (digest `b709a63cfd15`) with base d_1, bound to τ_1, whose result is a failure with a certificate: an input beginning an unterminated comment is accepted by τ_1 and rejected by the grammar. Certified admissibility fails, and the disposition is `CONTRADICTED`. The receipt ρ_1, indexed by $\kappa_1 =$ `d824bea6d198`, is appended to L. The head of H remains d_1.

Stale evidence. The proposer constructs τ_2 (digest `6ee6064ff948`) preserving the grammar, and obtains evidence ω_2 (digest `4abfdf5220f8`) against d_1 with footprint containing the tokenizer, the lexer table and both libraries. Before τ_2 is committed, the maintainer commits an unrelated authorized change e_2 that updates the lexer table, with its own evidence and authorization, giving $d_2 =$ `351c1718bc4d`. The submission of τ_2 with ω_2 is now evaluated against the head d_2. The event e_2 touches an element of the footprint of ω_2, so ω_2 is not applicable and the disposition is `STALE`, with e_2 as the intersecting event in the certificate. Receipt ρ_2 has $\kappa_2 =$ `87bb510b58bf`. The earlier refusal ρ_1 is unaffected and H is unchanged by either refusal.

Recomputation and commitment. The candidate is rebased on the new lexer table as τ_2' (digest `98e24356c56a`), and the verifier produces ω_3 (digest `18ed42459b75`) against d_2 with a passing result. The evidence is applicable, since it binds τ_2', the current specification and a base that is the head. The maintenance delta discharges obligations o_1 and o_2 and adds none, so M^+ falls from 70 to 40, which satisfies the budget rule in its first regime. The maintainer's authorization binds the principal, the transformation digest and the expected parent d_2, and is checked against Γ_{d_2}. The commit is performed atomically against the head, which is still d_2, so the commitment record shows equal expected and actual parents. Event e_3 is appended, giving $d_3 =$ `8c7168460ebf`.

Repair. A later input class, unterminated comments at the end of a file, is found to be mishandled, an escaped defect not covered by the first corpus. A repair sphere is opened, and its transformation and evidence pass certified admissibility. The repair event e_4 names d_3 as the repaired event, with the cause recorded, and is appended, giving $d_4 =$ `49799be056ad`. The digests d_1 to d_3 and the events e_1 to e_3 are unchanged, and the defective commit remains identifiable. The escaped defect also shows that the grammar corpus used by the checking procedure was inadequate. A conforming project would record a second repair, an amendment of the profile that extends the corpus, as a separate governance transaction under Section 10. The fixture records only the first repair.

The chain from d_1 to d_4, the four evidence applicability decisions, the three dispositions and the unchanged-head property of the refusals are all reproducible from the fixture by an independent kernel. A conforming kernel MUST arrive at the same digests and the same dispositions.

20 Conformance and Experimental Evaluation

20.1 Conformance suite

The conformance suite tests the properties on which every claim of the protocol depends. It checks deterministic replay, by replaying recorded histories on independent kernels and comparing final digests and derived states. It checks that commitments without valid authorization are rejected, that evidence computed against an outdated history is rejected when its footprint intersects intervening events and accepted when it does not, and that commitments violating a registered invariant are rejected. It checks that capabilities are enforced for each operation and that a computational principal cannot exercise a capability it was not granted. It checks the repair mechanism, confirming that repair events extend the history without altering earlier digests. It checks platform equivalence by running the same specification against several backends and confirming membership in the acceptable sets. It checks the handling of conflicting proposals that touch overlapping footprints, including the requirement that the second proposal receive a disposition that cites the first. It checks that two commitments constructed against the same head and conflicting in the sense of Proposition 8 are serialized so that the second is evaluated against the new head, that a governance amendment is evaluated against the pre-transition state, that distinct serializations of one event yield one digest, that disposition types are preserved in receipts, and that capacity deferrals are aggregated without loss of the count. The decisive test is a nested-sphere fixture. Two subordinate spheres independently pass every local check, and their composition violates an obligation of the parent. A conforming kernel MUST preserve both local certificates, refuse the composed transformation with a finding that identifies the violated boundary obligation, and leave the authoritative head unchanged. The worked example of Section 19 is the first fixture, the nested-sphere test is the second, and a small repair-target fixture is the third. All three are reproduced by the limited Python and Rust reference kernels of Section 18, which also agree step by step on randomly generated scenarios. The nested-sphere fixture includes a reference checker that computes the expected refusal for the exports example used in Proposition 9. Each test is stated as a pair of a recorded input history and an expected disposition, so that independent implementations can be compared mechanically.

20.2 Comparative study

Whether the protocol reduces the problems it was designed against is an empirical question. A controlled comparison would evaluate four conditions on matched sets of maintenance and feature tasks: conventional manual development, unrestricted assisted development, assisted development with conventional continuous integration, and assisted development governed by

the protocol. Measures would not include volume of generated code. They would include reviewer effort, escaped defects, architectural complexity, dependency growth, time to repair, reproducibility of builds, maintenance burden as recorded in the obligation registry and later revised, and the fraction of consequential decisions that were resolved by an explicit refinement and not by default.

The analysis plan, tasks and metrics should be fixed before data are collected. Tasks should be assigned to conditions by a paired design so that differences in task difficulty do not masquerade as effects of the protocol, reviewers should be blinded to condition where the artifacts allow it, and results should be reported as effect sizes with interval estimates and not as binary significance claims. The design should state in advance the outcomes that would count against the protocol, which include no reduction in reviewer effort, no reduction in escaped defects, or a reduction obtained only by refusing so many proposals that useful work stopped. A result of that kind would be informative and would be reported. Absent such a study, the protocol is a design and not a demonstrated improvement.

21 Limitations and Open Problems

Several limits should be stated plainly. The protocol adds recording and checking work, and for small projects with a single maintainer the overhead may exceed the benefit. The suitable profile for such projects is a reduced one, retaining the history, the disposition record and the separation of generation from commitment, and omitting budgets and multidimensional evaluation until they are needed.

Maintenance estimates are uncertain, and the protocol records the uncertainty without removing it. A budget enforced on poor estimates can block valuable work or admit costly work, and the mechanisms of revision and exception are the only protection against systematic error. Verifier independence cannot be established by declaration alone. A verifier written by the same author or trained on the same material as the proposer may share its defects, and the independence class is a record of what the project knows about this risk and not a guarantee against it.

The protocol relocates authority and does not remove the need to decide who holds it. Governance structures can be captured, can adopt weightings that suit some participants at the expense of others, and can restrict the roles available to people without institutional standing. Making these decisions explicit and reviewable is a precondition for contesting them. It does not settle them. For the same reason, the protocol offers no remedy for unequal access to the computation needed to run verifiers, backends or models.

Several technical questions are left open in version 0.1. Merging distributed histories requires a conflict semantics for events that commute only partly, and for authorization records issued under divergent governance states. Equivalence of refused proposals is undecidable in general and the fingerprint approximation needs empirical tuning. The best representation of footprints for evidence freshness is unsettled, since overly coarse footprints reject valid evidence and overly fine ones admit stale evidence. The repository implementation has not been audited for

compatibility with these definitions. Finally, the experimental program of Section 20 has not been carried out, and the claims of this specification about the benefits of the protocol should be treated as hypotheses until it has.

References

[1] Lattner, C. Interview on the Digital Disruption program concerning software engineering with generative systems, Mojo and the Modular platform. Recorded interview.

[2] Bradner, S. Key Words for Use in RFCs to Indicate Requirement Levels. RFC 2119, Internet Engineering Task Force, 1997.

[3] Strathern, M. “Improving Ratings”: Audit in the British University System. *European Review* 5(3), 305–321, 1997.

[4] Lattner, C. and Adve, V. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, 75–86, 2004.

[5] Rice, H. G. Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society* 74(2), 358–366, 1953.

[6] Chacon, S. and Straub, B. *Pro Git*, second edition. Apress, 2014.

[7] Rundgren, A., Jordan, B. and Erdtman, S. JSON Canonicalization Scheme (JCS). RFC 8785, Internet Engineering Task Force, 2020.

[8] Necula, G. C. Proof-Carrying Code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, 106–119, 1997.

[9] Meyer, B. Applying “Design by Contract”. *IEEE Computer* 25(10), 40–51, 1992.

[10] Jones, C. B. Tentative Steps Toward a Development Method for Interfering Programs. *ACM Transactions on Programming Languages and Systems* 5(4), 596–619, 1983.

[11] Misra, J. and Chandy, K. M. Proofs of Networks of Processes. *IEEE Transactions on Software Engineering* 7(4), 417–426, 1981.

[12] Dennis, J. B. and Van Horn, E. C. Programming Semantics for Multiprogrammed Computations. *Communications of the ACM* 9(3), 143–155, 1966.

[13] Kung, H. T. and Robinson, J. T. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems* 6(2), 213–226, 1981.

[14] Fowler, M. Event Sourcing. Essay on martinfowler.com, 2005.

[15] Torres-Arias, S., Afzali, H., Kuppusamy, T. K., Curtmola, R. and Cappos, J. in-toto: Providing Farm-to-Table Guarantees for Bits and Bytes. In *Proceedings of the 28th USENIX Security Symposium*, 1393–1410, 2019.

[16] Lamb, C. and Zacchiroli, S. Reproducible Builds: Increasing the Integrity of Software Supply Chains. *IEEE Software* 39(2), 62–70, 2022.

[17] Flyxion. *Distinction Under Constraint: Admissibility, Irreversibility, and Reconstruction in a Family of Research Frameworks*. Independent publication, November 2026. `https://standardgalactic.github.io/voice-cloning/distinction-under-constraint.pdf`.

[18] Flyxion. *Programs as Histories: Projection, Admissibility, and the Geometry of Computation*. Independent publication, June 2026. `https://standardgalactic.github.io/playfloor/working/programs-as-histories.pdf`.

[19] Flyxion. *Software Degradation and the Loss of Taste*. Independent publication. `https://standardgalactic.github.io/alignment/software-degradation.pdf`.