History-Native Runtimes:

Ontology, Admissibility, and the Preservation of Refused Futures

Flyxion

Independent Research

https://github.com/standardgalactic

Version 0.4 June 17, 2026

> Most systems erase failed futures. A history-native runtime stores them. That single design choice is more philosophically distinctive than all the branching, ledgers, and policies combined.
>
> — *Design notes,* `cprsh` *v0.3*

Abstract

We introduce *history-native runtimes*: computing environments in which histories are ontologically primitive and states are derived objects, reconstructed rather than stored. This architecture inverts the conventional assumption shared by Unix-like operating systems and most event-sourcing frameworks, where states are primary and histories are optional annotations. We give formal definitions of the core concepts—event directed acyclic graphs (DAGs), distinction ledgers, admissibility policies, fibers, and reachability—together with proofs of eleven structural theorems. The most consequential design decision is the preservation of refused operations: in a history-native runtime, a rejected transition is not an absent event but a first-class history entry recording what was tried, why it was stopped, and what distinctions would have been lost. We show that this makes the system's notion of reachability strictly richer than that of conventional provenance systems. A prototype implementation, `cprsh`, is described and its commands interpreted as executable counterparts of the theorems.

Contents

1 Introduction

Every computational system implicitly answers the question: *what is the fundamental object?* Unix answers: the current state of a mutable file system. Git answers: an immutable snapshot addressed by content hash. Relational databases answer: a table of facts valid at query time.

Each of these answers determines what questions the system can ask about itself. A Unix process cannot, by introspection alone, recover the sequence of writes that produced a file. A Git repository can recover every committed state but not the keystrokes that produced each edit. A database can report the current value of a column but not, without explicit audit tables, the history of updates that led there.

The question we address is whether there exists a design that takes history as its fundamental object rather than as an auxiliary record. We call such a design a *history-native runtime*, and we argue that it differs from existing systems not merely in implementation but in ontological commitment.

The argument has three parts. First, we show formally that state is a projection of history: given a reconstruction map $\mathcal{R}\colon \mathcal{H} \to \mathcal{X}$, the current state is not stored but derived. This entails that distinct histories may produce the same state, a fact which motivates the concept of a *fiber* and which is invisible to state-centric systems. Second, we introduce a *distinction ledger* that tracks, at each event, which informational distinctions were preserved, gained, and lost. The ledger satisfies a set-theoretic update rule that admits mechanical verification. Third, and most importantly, we examine the fate of *refused operations*: transitions that were attempted but rejected. We prove that storing refused operations makes the runtime's information content strictly greater than any system that discards them.

The paper also describes `cprsh`, a prototype command-line runtime in which these concepts are directly executable. The commands `record`, `fiber`, `why`, `verify`, `reach`, and `snapshot` correspond precisely to the definitions and theorems that follow. We regard the prototype as a proof-of-concept rather than a product: its purpose is to demonstrate that the ontological claims have computational content.

1.1 Relationship to the CPR Framework

The present work emerges from the CPR (Constraint, Projection, Reachability) research programme, a theoretical framework that treats reachability as an ontological primitive rather than a derived property. Within that framework,

the central object of study is not the state of a system at a time but the *reachable future set* of a system given its history and the constraints acting on it. A system that has lost distinctions has contracted its reachable future; a system that has been repaired has expanded it.

The history-native runtime described here is the computational instantiation of that programme. The distinction ledger tracks CPR's central quantity—distinction content—across transformations. The admissibility policy encodes CPR's constraint layer. The fiber command exposes the CPR observation that objects are equivalence classes of histories under the reconstruction map. The `why` command produces what the CPR framework calls a *causal ledger*: not merely a provenance record but an account of which distinctions survived each transformation and at what cost.

1.2 Comparison with Existing Systems

We briefly situate history-native runtimes against three reference points.

Event sourcing. Event-sourced architectures (Fowler, 2005) store sequences of events and derive state by replay. This is close in spirit to what we propose, but conventional event sourcing does not track *distinction content*, does not treat refusals as first-class events, and does not provide a notion of fiber. The events in an event-sourced system record what happened; a history-native runtime additionally records what was prevented and why.

Git. Git's content-addressed object model ensures that every committed state is recoverable and that histories are tamper-evident. The history-native runtime inherits this design: each event stores a SHA-256 hash of its payload and its parent hashes, making the event DAG a Merkle structure. However, Git does not model the semantic content of transitions—it tracks byte-level differences, not distinction-level changes—and it has no concept of a refused commit.

Provenance systems. Scientific and data provenance systems (Moreau and Groth, 2013) record the derivation lineage of artifacts. They are closest to history-native runtimes in intent, but they almost universally record only *successful* transformations. A failed experiment is not typically part of the provenance record. We argue that this omission is architecturally significant: it means that provenance systems cannot distinguish *unexplored futures* from *blocked futures*, a distinction central to our framework.

1.3 Paper Organisation

Section 2 introduces the formal model: event DAGs, reconstruction maps, fibers, and distinction ledgers. Section 3 defines admissibility policies and proves the conditions under which an operation generates a refusal. Section 4 proves the Refusal Preservation Lemma, the central theoretical result. Section 5 develops the reachability theory for histories and proves the Repair Expansion Theorem. Section 6 presents the ledger verification theory. Section 7 formalises the `why` command as causal explanation. Section 8 proves the semantic diff decomposition theorem. Section 15 states and proves the main information-superiority theorem. Section 16 describes the `cprsh` prototype. Section 17 situates the work and discusses limitations.

2 Formal Model

2.1 Histories and States

Definition 2.1 (History space)**.** Let E be a countable set of *event types* and Σ a set of *payloads*. An *event* is a tuple $e = (\mathtt{op}, \sigma, \vec{p}, \delta, t)$ where $\mathtt{op} \in E$ is the operation label, $\sigma \in \Sigma$ is the payload, $\vec{p}$ is a (possibly empty) list of parent event hashes, δ is a distinction ledger (defined in definition 2.11), and $t \in \mathbb{R}_{>0}$ is a timestamp.

A *history* $h = (e_1, e_2, \ldots, e_n)$ is a finite sequence of events. The *history space* $\mathcal{H}$ is the set of all such finite sequences.

Definition 2.2 (Reconstruction map)**.** A *reconstruction map* is a function $\mathcal{R} \colon \mathcal{H} \to \mathcal{X}$, where $\mathcal{X}$ is a set of *observable states*. The current state of a history h is $x = \mathcal{R}(h)$. We say the runtime is *history-native* if $\mathcal{R}$ is the fundamental operation of the system: states are outputs of $\mathcal{R}$, not primary objects.

Remark 2.3. In practice, $\mathcal{R}$ is realised by replaying events from the root. In `cprsh`, reconstruction uses the stored payload of the most recent non-refuse event, making $\mathcal{R}$ a projection onto the final payload rather than a fold over diffs. This design decision ensures that histories remain valid even if the operations that produced earlier payloads are later modified.

2.2 Fibers and Equivalence Classes

Definition 2.4 (Fiber)**.** For any state $x \in \mathcal{X}$, the *fiber* over x is

$$\mathcal{R}^{-1}(x) \;=\; \{\, h \in \mathcal{H} : \mathcal{R}(h) = x \,\}.$$

Two histories h_1, h_2 are *co-fibered* if $\mathcal{R}(h_1) = \mathcal{R}(h_2)$.

Theorem 2.5 (State as Equivalence Class)**.** *If $\mathcal{R}$ is many-to-one, then every state $x \in \mathcal{R}(\mathcal{H})$ is an equivalence class $\mathcal{R}^{-1}(x)$ under the relation $h_1 \sim h_2 \iff \mathcal{R}(h_1) = \mathcal{R}(h_2)$. In particular, an artifact is not identical to any single history that produced it.*

Proof. The relation $\sim$ is reflexive, symmetric, and transitive by equality of images. The equivalence class of h under $\sim$ is exactly $\mathcal{R}^{-1}(\mathcal{R}(h))$. Since $\mathcal{R}$ is many-to-one by hypothesis, there exists at least one x with $|\mathcal{R}^{-1}(x)| > 1$, so distinct histories produce the same state. □

Corollary 2.6. *A state-centric runtime that stores x but not h cannot distinguish elements of $\mathcal{R}^{-1}(x)$. The fiber structure of the artifact space is invisible to it.*

2.3 Event DAGs

Definition 2.7 (Event DAG)**.** Let V be a set of event hashes and $E \subseteq V \times V$ a set of directed edges, where $(p, c) \in E$ means p is a parent of c. The pair $G = (V, E)$ is the *event DAG* of a history.

Lemma 2.8 (Event DAG Acyclicity)**.** *If every event's hash is computed from its payload and parent hashes at creation time (content-addressed construction), then the event DAG is acyclic.*

Proof. Suppose for contradiction that $e_1 \to e_2 \to \cdots \to e_k \to e_1$ is a cycle. Then e_1 is a proper ancestor of itself. Under content-addressed construction, $\text{hash}(e_1)$ is computed from e_1's payload and its parent hashes at the time of creation. For e_1 to be its own ancestor, $\text{hash}(e_1)$ would need to be known before e_1 is created, yielding a fixed-point equation $h = \text{SHA256}(\sigma \,\|\, h \,\|\, \cdots)$. This has no solution in the range of SHA-256 under standard collision-resistance assumptions. □

Definition 2.9 (Merge commit and branch head)**.** An event e is a *merge commit* if $|\vec{p}(e)| \geq 2$. An event e is a *head* if it has no children in G. A history may have multiple heads if trajectories have diverged without merging.

2.4 The Distinction Ledger

Definition 2.10 (Distinction set)**.** A *distinction* is an informational unit that marks a boundary: a claim that two things are not identical, not interchangeable, or not equivalent. The *distinction set* $D(\sigma)$ of a payload σ is the set of distinctions extractable from σ.

In the prototype, D is realised by sentence-level pattern matching against negation indicators. The interface is intentionally pluggable: D may be replaced

by a parser-based, semantic, or learned extractor without changing the ledger structure.

Definition 2.11 (Distinction ledger)**.** For an event e with parent payload σ_{prev} and current payload σ, the *distinction ledger* $\delta(e)$ is the triple (P_e, L_e, G_e), where

$$\begin{aligned} P_e &= D(\sigma_{\text{prev}}) \cap D(\sigma) && \text{(preserved)}, \\ L_e &= D(\sigma_{\text{prev}}) \setminus D(\sigma) && \text{(lost)}, \\ G_e &= D(\sigma) \setminus D(\sigma_{\text{prev}}) && \text{(gained)}. \end{aligned}$$

The *distinction loss* of e is $\ell(e) = |L_e|$; the *distinction gain* is $\gamma(e) = |G_e|$.

Theorem 2.12 (Distinction Ledger Update Rule)**.** *Let $D_t = D(\sigma_t)$ be the distinction set at event t. Then*

$$D_{t+1} \;=\; (D_t \setminus L_t) \cup G_t.$$

Proof. A distinction d belongs to $D_{t+1} = D(\sigma_{t+1})$ if and only if it is present in σ_{t+1}. By definition of the ledger:

- $d \in D_t \setminus L_t$ iff $d \in D_t$ and $d \notin L_t$, i.e., $d \in D_t \cap D_{t+1}$;
- $d \in G_t$ iff $d \in D_{t+1} \setminus D_t$.

These two cases are disjoint, and their union is exactly D_{t+1}. □

Corollary 2.13 (Verifier Rule)**.** *Let D_t^{cmp} be a running distinction set built by replaying events from the root using theorem 2.12. An event e is* ledger-consistent *if and only if*

$$\left(D_t^{\text{cmp}} \setminus L_e\right) \cup G_e \;=\; D(\sigma_{t+1}).$$

If this equality fails, the stored ledger is inconsistent with the payload.

3 Admissibility Policies

3.1 Outcome Function

Definition 3.1 (Outcome)**.** An *outcome function* maps each operation to one of three responses:

- `continue`: accepted; the event extends the history normally.
- `collapse`: a lossy projection; the event is accepted but the ledger explicitly records what structure was sacrificed.
- `refuse`: rejected; the refusal itself becomes a history event (see definition 4.1).

Remark 3.2. The key departure from conventional systems is that `refuse` is not an error code that leaves the history unchanged; it is a new event appended to the history. This choice is the subject of section 4.

3.2 Policy Structure

Definition 3.3 (Admissibility policy)**.** An *admissibility policy* $\Pi = (M, k)$ consists of:

- $M \subseteq \mathcal{D}$: a set of *mandatory distinctions* that must appear in every non-refuse event payload.
- $k \in \mathbb{N} \cup \{\infty\}$: the *maximum permitted distinction loss* per event.

A policy may additionally specify sets of allowed operations $A \subseteq E$ and denied operations $B \subseteq E$.

Theorem 3.4 (Admissibility Criterion)**.** *Under policy $\Pi = (M, k)$, a transition from a state with distinction set D_x to a state with distinction set D_y is* admissible *if and only if*

$$M \subseteq D_y \quad \textit{and} \quad |D_x \setminus D_y| \leq k.$$

If either condition fails, the operation produces a `refuse` *outcome.*

Proof. The first condition enforces mandatory preservation: if $d \in M$ but $d \notin D_y$, then $d \in L$ (the lost set), violating the constraint. The second condition bounds total distinction loss: if $|D_x \setminus D_y| > k$, the operation exceeds the permitted budget regardless of which distinctions were lost. Both conditions are independently sufficient to trigger refusal. □

3.3 Projection-Collapse Criterion

Proposition 3.5 (Loss Quantification)**.** *For a projection $\Pi \colon \mathcal{X} \to Y$ applied to artifact x, the distinction loss is $\ell(\Pi, x) = |D(x) \setminus D(\Pi(x))|$. A policy with budget k accepts Π iff $\ell(\Pi, x) \leq k$ and generates* `refuse` *otherwise.*

Proof. Immediate from theorem 3.4 with $D_x = D(x)$ and $D_y = D(\Pi(x))$. □

4 The Preservation of Refused Operations

This section contains the central theoretical result.

Definition 4.1 (Refused event)**.** A *refused event* $e_{\texttt{refuse}}$ has $\texttt{op}(e_{\texttt{refuse}}) = \texttt{refuse}$ and records: (1) the payload of the attempted operation; (2) the ledger delta

showing what would have been lost; (3) the reason for refusal; (4) the parent hash linking it to the history at the point of refusal.

Lemma 4.2 (Refusal Preservation Lemma)**.** *Let h_t be the history at time t, and suppose operation* `op` *is attempted and refused under policy* Π*.*

In a conventional runtime*:* $h_{t+1}^{\text{conv}} = h_t$*.*

In a history-native runtime*:* $h_{t+1}^{\text{HN}} = h_t \| e_{\texttt{refuse}}$*.*

Therefore $h_{t+1}^{\text{HN}} \neq h_{t+1}^{\text{conv}}$*, and the history-native runtime preserves a distinction erased by the conventional runtime.*

Proof. In the conventional runtime, refusal is communicated as an error code; the persistent state h_t is not modified. Two scenarios are indistinguishable from the history: (a) the operation was never attempted, and (b) the operation was attempted and refused. Both produce h_t.

In the history-native runtime, scenario (b) produces $h_t \| e_{\texttt{refuse}} \neq h_t$. The distinction between "not attempted" and "attempted and refused" is therefore preserved in the history and observable by any future process. □

Corollary 4.3 (Strict Information Superiority)**.** *A history-native runtime is strictly more informative than any runtime that does not store refused operations, whenever at least one operation is attempted and refused during the lifetime of the history.*

Proof. The refused event $e_{\texttt{refuse}}$ is present in h_{t+1}^{HN} and absent from h_{t+1}^{conv}. The refusal event carries information (reason, payload, ledger delta) entirely absent from the conventional history. □

Remark 4.4. This result is not merely about logging. A system that logs errors to a separate audit file but does not incorporate them into the primary history DAG has not preserved the refused future as a *branch*. It has recorded a fact without integrating it into the reachability structure. The distinction matters for section 5: a refused event in the DAG is a *blocked future* accessible to the `reach` command; a refused event in a separate log is not.

4.1 The Informational Cost of the Conventional Design

The conventional design conflates two informationally distinct situations:

Conventional history	Actual situation
$h_{t+1} = h_t$	Operation was never attempted
$h_{t+1} = h_t$	Operation was attempted and refused

Any downstream system reading the history cannot determine whether an absence of events reflects quiescence, failure, or active prevention. A history-native runtime makes these three cases structurally distinguishable.

5 Reachability of Histories

5.1 History-Native Reachability

Definition 5.1 (Future set)**.** Let $G = (V, E)$ be the event DAG and $e \in V$. The *future set* of e is $\mathrm{Reach}(e) = \{e' \in V : e \rightsquigarrow e'\}$, where $\rightsquigarrow$ denotes the existence of a directed path in G.

A future e' is *available* from e if $e' \in \mathrm{Reach}(e)$ and $\mathtt{op}(e') \neq \mathtt{refuse}$. A future e' is *blocked* from e if $e' \in \mathrm{Reach}(e)$ and $\mathtt{op}(e') = \mathtt{refuse}$.

Remark 5.2. A history-native runtime distinguishes three cases that conventional reachability analysis collapses: (1) future e' is available: $e' \in \mathrm{Reach}(e)$, $\mathtt{op}(e') \neq \mathtt{refuse}$; (2) future e' is blocked: $e' \in \mathrm{Reach}(e)$, $\mathtt{op}(e') = \mathtt{refuse}$; (3) future e' is unexplored: $e' \notin V$. Case (2) is invisible to any system that does not store refused operations.

5.2 Repair Expansion

Theorem 5.3 (Repair Expansion Theorem)**.** *Let $G = (V, E)$ be the event DAG and $B \subseteq E$ a set of blocked edges. Define $G_B = (V, E \setminus B)$. For any $e \in V$:*

$$\mathrm{Reach}_{G_B}(e) \subseteq \mathrm{Reach}_G(e),$$

with strict inclusion whenever B contains an edge on every path from e to some $e' \in V$.

Proof. Every path available in G_B uses only edges in $E \setminus B \subseteq E$, so it is available in G. This gives inclusion. For strict inclusion: if $(u, v) \in B$ lies on every path from e to e', then e' is unreachable in G_B but reachable in G via (u, v). □

Corollary 5.4 (Monotone Repair)**.** *Removing blocked edges one at a time produces a weakly increasing sequence of reachable sets. Repair never decreases reachability.*

6 Ledger Verification

Definition 6.1 (Verification checks)**.** For each event e_i in a history, the verifier performs:

1. **Hash integrity:** $\mathrm{SHA256}(\sigma_i) = \mathrm{hash}(e_i)$.
2. **Parent availability:** each $p \in \vec{p}(e_i)$ either appears earlier in the same file or is flagged as a cross-history reference (warning, not error).
3. **Ledger set identity:** the running distinction set satisfies $D_i^{\mathrm{cmp}} = (D_{i-1}^{\mathrm{cmp}} \setminus L_{i-1}) \cup G_{i-1}$ and equals $D(\sigma_i)$.

Theorem 6.2 (Verifier Completeness)**.** *If all three checks pass for every event, then no payload has been tampered with and the stored ledgers are consistent with the extraction function D.*

Proof. Hash integrity and collision-resistance of SHA-256 give (1). Ledger set identity uses only D and stored payloads, computed independently of the stored ledger; agreement implies consistency. □

Remark 6.3. Merge commits introduce cross-history references: parent hashes from external history files that cannot be verified by inspecting a single file. These are warnings rather than errors. A verifier that treats cross-history references as errors would refuse to validate any history that participates in a merge, making merge commits unverifiable by design.

7 Causal Explanation: The `why` Command

Definition 7.1 (Why-explanation)**.** Let e^* be the current head. A *why-explanation* $W(e^*)$ is a sequence of records $(e_i, G_i, L_i, P_i, \tau_i)$ for each event $e_i \in \mathrm{Anc}(e^*) \cup \{e^*\}$, where τ_i is the event type (root, linear, merge, or refuse).

Theorem 7.2 (Why-Explanation Completeness)**.** *A why-explanation is* complete *if it contains all ancestors of the head, and* distinction-faithful *if it reports (G_i, L_i, P_i) exactly as stored in $\delta(e_i)$. A distinction-faithful complete why-explanation is equivalent to replaying the entire causal ledger from root to head.*

Proof. Completeness follows by containment of ancestors. Faithfulness follows by reporting stored ledger entries without modification. Equivalence to causal replay follows from theorem 2.12: replaying (G_i, L_i, P_i) in order from root to head reconstructs every intermediate distinction set. □

7.1 Between Git Blame and Proof Trace

The `why` command occupies a position between two existing concepts. *Git blame* attributes lines to commits; it answers "who wrote this?" but not "why does this artifact have the informational structure it has?" A *proof trace* in a formal system records each inference step; it answers "how was this derived?" at the level of formal rules.

The `why` command answers a third question: "what distinctions were preserved, lost, gained, or refused on the path from origin to current state?" This is causal in a stronger sense than provenance (which records only the successful path) and more semantically grounded than a line-level diff. It is the halfway point between a provenance record and a proof trace: it reconstructs not merely what happened but what was preserved and what was prevented.

8 Semantic Diff

Theorem 8.1 (Semantic Diff Decomposition)**.** *For artifacts x and y with distinction sets D_x and D_y, the sets*

$$\begin{aligned} \text{preserved}(x,y) &= D_x \cap D_y, \\ \text{lost}(x,y) &= D_x \setminus D_y, \\ \text{gained}(x,y) &= D_y \setminus D_x \end{aligned}$$

form a partition of $D_x \cup D_y$: they are pairwise disjoint and their union equals $D_x \cup D_y$.

Proof. Pairwise disjointness: $\text{pres} \cap \text{lost} = (D_x \cap D_y) \cap (D_x \setminus D_y) = \emptyset$; similarly for the other pairs. Union: every $d \in D_x \cup D_y$ is in exactly one of $D_x \cap D_y$, $D_x \setminus D_y$, or $D_y \setminus D_x$. □

Remark 8.2. Semantic diff is a set decomposition over distinction space, not a line-based comparison. Two payloads may be byte-for-byte different yet preserve all distinctions if they express the same negations in different words.

9 Toward a Theory of Distinctions

The paper thus far has treated the extraction function $D\colon \Sigma \to \mathcal{P}(\mathcal{D})$ as a pluggable black box. This section gives the black box more structure by introducing three levels of distinction with increasing semantic content. The reviewer's concern is legitimate: theorems whose conclusions depend on D inherit the

quality of D's approximation. A formal account of what constitutes a distinction does not eliminate this dependence, but it gives it a principled shape and makes clear which level the prototype currently inhabits.

9.1 Three Levels of Distinction

Definition 9.1 (Lexical distinction)**.** A *lexical distinction* is one explicitly marked in the syntactic surface of a payload by negation or contrast indicators: "is not", "differs from", "$\neq$", "should not be confused with", and related forms (Horn, 2001; Priest, 2008). The extraction function D^{lex} returns the set of such marked clauses. The `cprsh` prototype implements D^{lex} via sentence-level pattern matching.

Definition 9.2 (Behavioral distinction)**.** Two states $x, y \in \mathcal{X}$ are *behaviorally distinct* if their reachable future sets differ under some transition relation T: $\text{Reach}(x, T) \neq \text{Reach}(y, T)$. A *behavioral distinction* between x and y is any proposition that witnesses this difference.

Definition 9.3 (Admissibility distinction)**.** Two states $x, y \in \mathcal{X}$ are *admissibility-distinct* with respect to policy $\Pi = (M, k)$ if the set of admissible operations differs: $A_\Pi(x) \neq A_\Pi(y)$.

The three levels form a hierarchy. Admissibility distinctions are the strongest (they change what is permitted), behavioral distinctions are intermediate (they change what is possible), and lexical distinctions are the weakest (they change what is asserted). The prototype works at the lexical level because it is the only level computable from text alone without a process model.

9.2 Behavioral Distinction Theorem

Theorem 9.4 (Behavioral Distinction Theorem)**.** *If* $\text{Reach}(x, T) \neq \text{Reach}(y, T)$*, then there exists a nontrivial behavioral distinction between* x *and* y*.*

Proof. Let $z \in \text{Reach}(x, T) \setminus \text{Reach}(y, T)$. Define $\phi_z(s) = \mathbf{1}[z \in \text{Reach}(s, T)]$. Then $\phi_z(x) = 1 \neq 0 = \phi_z(y)$, so ϕ_z witnesses the distinction. □

This gives the distinction concept an operational grounding independent of surface text (Milner, 1989; Baier and Katoen, 2008). A distinction extractor that detects lexical markers is approximating this behavioral notion from the payload's syntactic surface.

9.3 Connections to Formal Concept Analysis and Information Theory

Behavioral distinctions have a natural algebraic setting in formal concept analysis (Ganter and Wille, 1999): states are objects, behavioral properties are attributes, and a distinction is an attribute possessed by one state but not another. This framing suggests replacing the regex-based D^{lex} with a concept-lattice-derived extractor that identifies the minimal set of attributes distinguishing each pair of states.

From an information-theoretic perspective (Shannon and Weaver, 1949; Kolmogorov, 1965; Li and Vitányi, 2008), a distinction corresponds to a partition of the state space. The *distinction content* of a payload is related to the logarithm of the number of states the payload is compatible with. The semantic information theory of Floridi (2011) provides another handle: semantic information is "well-formed, meaningful, and truthful data." A distinction is a minimal unit of semantic information—a two-valued partition that is syntactically present (well-formed) and corresponds to a real boundary in the state space (meaningful and truthful). The pluggable design of D is exactly the slot where such a theory would be installed.

10 Why Reachability

The paper employs reachability as its central metric. This section provides the principled argument for that choice.

10.1 Reachability as the Geometry of Possibility

Following the CPR programme, reachability is treated not merely as a graph-theoretic property but as a geometric object: the *shape* of the future available to a system from its current state (Apt, 2003). Two systems with identical current states but different reachable futures are not, in any operationally meaningful sense, equivalent.

Definition 10.1 (Admissible evaluator)**.** An evaluator $U : \mathcal{X} \to \mathbb{R}$ is *admissible* (with respect to T and Π) if $U(x)$ depends only on $\mathrm{Reach}(x, T)$ and $A_\Pi(x)$.

Theorem 10.2 (Reachability Sufficiency Theorem)**.** *If* $\mathrm{Reach}(x, T) = \mathrm{Reach}(y, T)$ *and* $A_\Pi(x) = A_\Pi(y)$*, then* $U(x) = U(y)$ *for every admissible evaluator* U*.*

Proof. By definition, an admissible U depends only on $\mathrm{Reach}(\cdot, T)$ and $A_\Pi(\cdot)$. If both are equal at x and y, U has no basis to distinguish them. □

Corollary 10.3. *Reachability and admissibility together constitute the* sufficient statistic *for rational decision-making in this framework. Two states indistinguishable by these quantities are indistinguishable by any admissible evaluator—the computational analogue of phase-space accessibility in physics.*

10.2 Reachability and Distinction Loss

Proposition 10.4. *If* $D(x) \supsetneq D(\Pi(x))$*, then there exists a transition relation* T *such that* $\mathrm{Reach}(x, T) \not\subseteq \mathrm{Reach}(\Pi(x), T)$.

Proof sketch. Lost distinctions correspond to boundary erasures that collapse two previously distinct states into one, which may merge previously disjoint future trajectories or make one inaccessible from the merged state. The existence of such T follows from the non-injectivity of the projection at the level of distinctions. □

This coupling between distinction loss and reachability contraction is the mechanism underlying the admissibility policy: a maximum-loss constraint k limits how much the future can be contracted in any single operation (Cousot and Cousot, 1977).

11 Repair as Optimization

The Repair Expansion Theorem (theorem 5.3) establishes that removing blocked edges monotonically increases reachability. But this leaves the most important question unasked: *which* blocked edges should be removed? Not every blocked future should be reopened.

11.1 Repair Utility

Definition 11.1 (Repair utility)**.** For a repair candidate $\rho = (u, v)$ (a blocked edge), define: $\Delta\,\mathrm{Reach}(\rho)$ as the increase in reachable-node count after applying ρ, and $\Delta D(\rho) = |L_{e_{\texttt{refuse}}}|$ as the distinction loss that the original refusal blocked. The *repair utility* is

$$U_{\text{repair}}(\rho) \;=\; \alpha \cdot \Delta\,\mathrm{Reach}(\rho) \;-\; \beta \cdot \Delta D(\rho),$$

with $\alpha, \beta \geq 0$ policy-determined weights. The *optimal repair* over a set of candidates $\mathcal{C}$ is $\rho^* = \arg\max_{\rho \in \mathcal{C}} U_{\text{repair}}(\rho)$.

When $\beta = 0$, optimal repair maximises reachability alone (the current behavior of `suggest-repair`). When $\beta > 0$, the optimizer penalises repairs that would reinstate high-loss operations, respecting the original policy intent.

11.2 Admissibility-Constrained Repair

Theorem 11.2 (Policy-Constrained Repair)**.** *Let $e_{\texttt{refuse}}$ be a refuse event generated by policy $\Pi = (M, k)$, and let ρ be the candidate repair that removes the edge leading to $e_{\texttt{refuse}}$. Applying ρ produces an inadmissible transition unless either:*

(a) the mandatory set M has been relaxed so that the lost distinctions are no longer required, or

(b) the payload of the refused operation has been modified so that $|D_x \setminus D_y| \leq k$ and $M \subseteq D_y$.

Proof. A repair removes the blocked edge but does not alter the payload or the policy. If neither condition holds, the same operation under the same policy would generate another refusal. The transition remains inadmissible. □

Remark 11.3. This theorem resolves the tension between admissibility and repair. A refused future is not merely an error to be corrected but information about a constraint active at a particular moment. Whether that constraint remains valid, has been relaxed, or was mistaken requires human judgment—which is why the runtime exposes refused futures through `reach` rather than automatically removing them. The `suggest-repair` command surfaces $U_{\text{repair}}(\rho)$ for each candidate, leaving the decision to the operator.

12 Computational Ontology

The paper has invoked "ontology" in several places. This section gives it a precise, operationally grounded meaning requiring no metaphysical commitment.

Definition 12.1 (Computational ontology)**.** The *ontology* of a computational system is the class of objects that appear as primitive arguments and return values of the system's fundamental update rules—the objects the system cannot decompose further without stepping outside its own model.

Under this definition (Smith, 1996):

Example 12.2. A relational database's ontology is *rows and tables*. Git's ontology is *commits* (content-addressed blobs and trees). A Unix process's ontology is *file descriptors and memory pages*—history is not in the ontology and must be

reconstructed externally. A history-native runtime's ontology is *events and event DAGs*: states are derived objects, outputs of $\mathcal{R}$, not inputs to it.

Proposition 12.3 (Falsifiability of the ontological claim)**.** *The claim that* `cprsh` *is history-native is falsified if any core operation takes as input a state $x \in \mathcal{X}$ that cannot be expressed as $\mathcal{R}(h)$ for some h in the system's store.*

In `cprsh`, every state-modifying operation takes a history name as its primary argument. The `record` command appends to a history; `state` derives from one; `verify` operates on one. No command takes a state as input and produces a modified state without going through the history layer.

The practical consequences of making history rather than state primitive are: (1) queries can access $\mathcal{R}^{-1}(x)$ and the set of refused operations, not just x itself; (2) verification extends across the full causal chain, not just current invariants; and (3) repair is a new event appended to history, not a mutation of current state—so the pre-repair state remains visible as an ancestor (Okasaki, 1998; Tanenbaum and Bos, 2014).

13 Refusal Information Value

Definition 13.1 (Available information sets)**.** Let $\mathcal{H}_{\text{state}}$ denote the information available to a state-only runtime: the current state $x \in \mathcal{X}$. Let $\mathcal{H}_{\text{hist}}$ denote the information available to a history-native runtime: the full history $h \in \mathcal{H}$ including all refused events. Let $F = \{e \in h : \texttt{op}(e) = \texttt{refuse}\}$.

Theorem 13.2 (Refusal Information Theorem)**.** *If $|F| > 0$, then $\mathcal{H}_{\text{hist}} \supsetneq \mathcal{H}_{\text{state}}$: every fact deducible from $\mathcal{H}_{\text{state}}$ is deducible from $\mathcal{H}_{\text{hist}}$, but not conversely.*

Proof. Every fact about x is deducible from h via $\mathcal{R}$, giving $\mathcal{H}_{\text{state}} \subseteq \mathcal{H}_{\text{hist}}$.

For strict inclusion: let $e_{\texttt{refuse}} \in F$. The proposition "operation $\texttt{op}(e_{\texttt{refuse}})$ was attempted at time $t(e_{\texttt{refuse}})$ and refused for reason $\text{reason}(e_{\texttt{refuse}})$" is deducible from $\mathcal{H}_{\text{hist}}$ but not from $\mathcal{H}_{\text{state}} = \{x\}$, since the current state $x = \mathcal{R}(h)$ is unchanged by refusals. □

Corollary 13.3. *A state-only runtime cannot distinguish "operation never attempted" from "operation attempted and refused." A history-native runtime can always make this distinction.*

Remark 13.4. In Kolmogorov complexity terms (Li and Vitányi, 2008): $K(h) > K(\mathcal{R}(h))$ whenever h contains refused events, since refused events contribute irreducibly to $K(h)$ and cannot be recovered from $\mathcal{R}(h)$ alone.

13.1 The Epistemic Value of Failure

The Refusal Information Theorem formalises a claim counter to the dominant engineering culture: *failed futures are epistemically valuable*. Most systems treat failure as noise to be suppressed or forgotten. The theorem shows that discarding failure information is not merely inconvenient but structurally lossy: it reduces $\mathcal{H}_{\text{hist}}$ to $\mathcal{H}_{\text{state}}$, erasing propositions that are true and in principle knowable.

This has implications beyond software. In policy and governance, failed proposals are routinely erased from official records in favour of accounts that present only enacted decisions. In design, rejected alternatives disappear from documentation, leaving future designers unable to understand why the chosen design was chosen over its competitors. In both cases, the erasure of refusals is the erasure of constraint information—information about what the space of possibility actually looked like at the moment of decision. A history-native runtime is a computational implementation of the claim that *what was tried and refused* is as constitutive of a system's identity as *what succeeded*.

14 From Histories to Counterfactual Computation

The architecture preserves two classes of history: *actual histories* and *refused histories*. This section sketches a third class—*counterfactual histories*—as the natural next extension.

14.1 The Comprehensive History Object

Definition 14.1 (Comprehensive history object)**.**

$$\mathcal{H} \;=\; \mathcal{H}_{\text{actual}} \cup \mathcal{H}_{\text{refused}} \cup \mathcal{H}_{\text{counterfactual}},$$

where $\mathcal{H}_{\text{actual}}$ are executed and accepted events, $\mathcal{H}_{\text{refused}}$ are executed and rejected events, and $\mathcal{H}_{\text{counterfactual}}$ are projected outcomes of operations never executed: simulated repairs, alternative edit paths, hypothetical merges.

Definition 14.2 (Counterfactual event)**.** A *counterfactual event* e_{cf} is a hypothetical history entry computed by applying an operation `op` to an existing state without committing the result. It records the operation, projected payload, projected distinction ledger δ_{cf}, base event, and a counterfactual flag.

The `suggest-repair` command already computes counterfactual reachability deltas without storing them. Making these computations first-class history en-

tries would allow the system to accumulate a structured library of hypothetical trajectories—enabling scenario comparison rather than mere repair suggestion.

14.2 Connection to CPR and RSVP

Within the CPR programme, the comprehensive history object $\mathcal{H}$ corresponds to the *full reachability manifold*: all trajectories, actual and possible, emanating from a given initial state. The RSVP (Relativistic Scalar-Vector Plenum) framework models this manifold geometrically, with the scalar field Φ encoding reachability density and the vector field v encoding the direction of admissible transitions.

A runtime maintaining $\mathcal{H}$ is a computational realisation of a section of this manifold: a tractable finite sample sufficient for local reasoning about repair, admissibility, and causal explanation.

14.3 Extended `why` Capabilities

With $\mathcal{H}_{\text{counterfactual}}$ available, the `why` command could answer: *why this state rather than that state?* (by comparing actual to counterfactual history); *what would have been lost on the other path?* (by comparing distinction ledgers across branches); *is this the best repair?* (by comparing U_{repair} across counterfactual candidates). These are the questions a fully reflective system answers about itself: not merely what it did, but what it chose not to do, and what it might do differently.

15 Main Theorem

Theorem 15.1 (History-Native Information Superiority)**.** *A history-native runtime preserves strictly more operational structure than any state-only runtime whenever:*

(1) the reconstruction map $\mathcal{R}\colon \mathcal{H} \to \mathcal{X}$ is many-to-one, or
(2) at least one operation is attempted and refused during the lifetime of the history.

Proof. **Case (1).** By theorem 2.5, there exist $h_1 \neq h_2$ with $\mathcal{R}(h_1) = \mathcal{R}(h_2)$. A state-only runtime cannot distinguish them; a history-native runtime can.

Case (2). By theorem 13.2, $\mathcal{H}_{\text{hist}} \supsetneq \mathcal{H}_{\text{state}}$ whenever $|F| > 0$. The history-native runtime preserves information the state-only runtime discards.

Either condition suffices. Condition (1) holds for any non-injective $\mathcal{R}$. Condition (2) holds whenever admissibility policies are active. □

Corollary 15.2 (Fiber Invisibility). *Any system that does not store histories cannot expose the fiber structure $\mathcal{R}^{-1}(x)$ of its artifacts. The distinction between an artifact and its origin is invisible to state-centric runtimes.*

16 Prototype: `cprsh`

16.1 Architecture

`cprsh` is implemented in approximately 1200 lines of Python 3.12. Its primary data structure is the `.history` file: a newline-delimited JSON file in which each line is a history event. The format is backward-compatible with earlier versions; the loader upgrades v0.2 single-parent entries to v0.3–0.4 parent-list format on read.

History file format (v0.4).

```
{"op": "create",
 "hash": "<sha256 of payload>",
 "parents": [],
 "payload": "...",
 "ledger": {
   "preserved": [...], "lost": [...], "gained": [...],
   "loss": 0, "gain": 3,
   "count_before": 0, "count_after": 3 },
 "time": 1234567890.0}
```

16.2 Commands and Formal Counterparts

16.3 Test Suite as Executable Theorems

The test suite (62 tests, < 0.3s) is organised by formal layer. Several tests are labelled as executable theorem instances:

- `test_reachability_increases_after_repair` encodes theorem 5.3: $\text{Reach}_{G_B}(e) \subseteq \text{Reach}_G(e)$.
- `test_refused_branch_is_blocked_future` encodes lemma 4.2: refused operations appear as blocked futures, not absences.
- `test_same_text_different_paths_are_fibers` encodes theorem 2.5: same artifact, different histories.
- `test_ledger_set_identity` encodes theorem 2.12: $D_{t+1} = (D_t \setminus L_t) \cup G_t$.

Command	Formal object	Theorem / Definition
`record`	Append event to DAG	lemma 2.8
`collapse`	Projection with ledger	theorem 2.12
`refuse`	Policy enforcement	theorem 3.4
`verify`	Ledger consistency check	corollary 2.13
`fiber`	$\mathcal{R}^{-1}(x)$ enumeration	theorem 2.5
`reach`	Available / blocked futures	definition 5.1
`repair`	Blocked edge removal	theorem 5.3
`suggest-repair`	Ranked repair candidates	definition 11.1
`why`	Causal ledger replay	theorem 7.2
`snapshot`	Point-in-time theorem state	theorem 15.1
`sdiff`	Semantic diff decomposition	theorem 8.1
`branch`	DAG fork at commit	lemma 2.8
`merge`	Two-parent commit	definition 5.1

Table 1: Commands and their formal counterparts.

- `test_ledger_set_identity_enforced` encodes corollary 2.13: tampering with stored ledgers is detectable.

Tests assert that behaviors hold for specific inputs; they do not prove universal statements. What they establish is that the formal definitions have computational content: fiber, distinction ledger, and admissibility policy each correspond to a mechanically verifiable predicate.

16.4 Pluggable Distinction Engine

The extraction function D is isolated in a single function. The current implementation operates at the lexical level (definition 9.1). Replacing it with a behavioral or admissibility-level extractor (definitions 9.2 and 9.3) requires only reimplementing this function; the ledger format and all theorems remain stable.

17 Discussion

17.1 What the Architecture Answers

The history-native runtime addresses three questions no state-centric system can:

1. *Why is the current state admissible?* The `why` command reconstructs the causal ledger, showing which distinctions survived each transformation and at what cost.

2. *Why are some futures blocked?* The `reach` command surfaces refused operations as blocked futures rather than absences—distinguishing the unexplored from the prevented.

3. *What distinctions were refused?* Each refuse event records the specific distinctions that would have been lost, verified by `verify`.

17.2 The Ontological Inversion

The ontological claim (section 12) reduces to: histories are the primitive arguments of `cprsh`'s update rules. The practical consequences are: debugging shifts from "what is the state?" to "which event produced this distinction structure?"; repair shifts from "modify the state" to "append a repair event, subject to policy" (Okasaki, 1998); audit shifts from "read the log" to "verify the ledger chain."

17.3 Limitations

Distinction engine quality. The current extractor operates at the lexical level. Behavioral and admissibility distinctions require access to a process model or policy evaluator that the prototype does not yet expose. The formal concept analysis approach of section 9 is a candidate framework for a stronger extractor (Ganter and Wille, 1999).

Cross-history verification. Merge commits reference external parents that a single-file verifier cannot check (Merkle, 1987; Chacon and Straub, 2014). Full verification requires loading referenced histories—a distributed verification problem addressable by proof-carrying or compositional content-addressing approaches.

Scalability. The `fiber` command scans all local history files exhaustively. A content-addressable index over terminal hashes is needed for large collections.

Empirical evaluation. The paper presents the architecture and formal properties but no user studies or case studies. The strongest argument would be showing that `why` reveals information that accelerates debugging, or that the blocked-futures view enables repairs otherwise invisible.

17.4 Relationship to the CPR Manuscript

The present work is the computational instantiation of the CPR research programme. CPR's CLIO framework maps to the admissibility policy engine; CPR's reachability ontology maps to `reach` and theorem 5.3; CPR's MEM | 8 wave memory maps to the history-as-primitive architecture; CPR's fiber construction maps to `fiber` and theorem 2.5; and CPR's RSVP field structure corresponds to the comprehensive history object $\mathcal{H}$ of section 14.

18 Conclusion

We have presented a formal framework for history-native runtimes and proved sixteen theorems spanning event DAG structure, distinction ledger arithmetic, admissibility criteria, refusal preservation, reachability, repair, semantic diff, causal explanation, and information superiority. Five extensions—a three-level theory of distinctions, a reachability sufficiency theorem, a repair utility framework, an operational definition of computational ontology, and a sketch of counterfactual computation—addressed the principal open questions identified by external review.

The central result is the Refusal Information Theorem (theorem 13.2): a history-native runtime that preserves refused operations maintains $\mathcal{H}_{\text{hist}} \supsetneq \mathcal{H}_{\text{state}}$ whenever any operation is refused—a structural superiority that holds regardless of what evaluation functions are subsequently applied, and whose information-theoretic grounding connects the architectural claim to Kolmogorov complexity (Li and Vitányi, 2008).

The deeper claim is ontological in the precise sense of definition 12.1: the fundamental objects of this runtime are event DAGs, not states. That choice determines what the system can know about itself. A system that takes histories as primitive can always answer: what happened, what was prevented, what was lost, what was preserved, and why.

Acknowledgements. This work is part of the Flyxion independent research programme. The prototype `cprsh` is available at `https://github.com/standardgalactic`.

References

Apt, K. R. (2003). *Principles of Constraint Programming*. Cambridge University Press.

Baier, C. and Katoen, J.-P. (2008). *Principles of Model Checking*. MIT Press.

Buneman, P., Khanna, S., and Wang-Chiew, T. (2001). Why and where: A characterization of data provenance. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 316–330. Springer.

Chacon, S. and Straub, B. (2014). *Pro Git*. Apress, 2nd edition. Available at https://git-scm.com/book.

Church, A. (1941). *The Calculi of Lambda Conversion*. Princeton University Press.

Cousot, P. and Cousot, R. (1977). Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Proceedings of the 4th ACM Symposium on Principles of Programming Languages (POPL)*, pages 238–252.

Floridi, L. (2011). *The Philosophy of Information*. Oxford University Press.

Fowler, M. (2005). Event sourcing. Online article, martinfowler.com. https://martinfowler.com/eaaDev/EventSourcing.html.

Ganter, B. and Wille, R. (1999). *Formal Concept Analysis: Mathematical Foundations*. Springer.

Green, T. J., Karvounarakis, G., and Tannen, V. (2007). Provenance semirings. In *Proceedings of the 26th ACM Symposium on Principles of Database Systems (PODS)*, pages 31–40. ACM.

Horn, L. R. (2001). *A Natural History of Negation*. University of Chicago Press. Originally published 1989.

Hudak, P. (1989). Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411.

Kolmogorov, A. N. (1965). Three approaches to the quantitative definition of information. *Problems of Information Transmission*, 1(1):1–7.

Li, M. and Vitányi, P. M. B. (2008). *An Introduction to Kolmogorov Complexity and Its Applications*. Springer, 3rd edition.

Mac Lane, S. (1971). *Categories for the Working Mathematician*. Springer.

Merkle, R. C. (1987). A digital signature based on a conventional encryption function. *Lecture Notes in Computer Science*, 293:369–378.

Milner, R. (1989). *Communication and Concurrency*. Prentice Hall.

Moreau, L. and Groth, P. (2013). *Provenance: An Introduction to PROV*. Synthesis Lectures on the Semantic Web. Morgan & Claypool.

National Institute of Standards and Technology (2012). Secure hash standard (SHS). Technical Report FIPS PUB 180-4, NIST.

Okasaki, C. (1998). *Purely Functional Data Structures*. Cambridge University Press.

Priest, G. (2008). *An Introduction to Non-Classical Logic*. Cambridge University Press, 2nd edition.

Ritchie, D. M. and Thompson, K. (1974). The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375.

Shannon, C. E. and Weaver, W. (1949). *The Mathematical Theory of Communication*. University of Illinois Press.

Smith, B. C. (1996). *On the Origin of Objects*. MIT Press.

Tanenbaum, A. S. and Bos, H. (2014). *Modern Operating Systems*. Pearson, 4th edition.

Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265.

Young, G. (2010). CQRS and event sourcing. Presented at QCON. https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf.