Provenance Algebra for Deterministic AI Execution

Replay Semantics for Stochastic Program Generators

Thomas Dionysopoulos, CFA

Abstract

When AI systems generate computational pipelines from natural language, provenance—the record of what was executed and why it is reproducible—is typically treated as execution metadata: logs, run IDs, artifact URIs. This paper argues that provenance for deterministic execution systems is not metadata but a semantic factorization of execution identity.

Building on prior work that established admissibility boundaries [1], canonical execution semantics [2], and quotient execution categories [3], we define a provenance map P_R over the quotient execution category $\mathcal{B}_R/\sim_R$, assigning to each execution equivalence class an 8-layer hash record that decomposes execution identity into semantic dependency boundaries. The provenance of a composed execution is determined by the provenance of its parts together with the declared dependency map between them: $P_R([g \circ f]_R) = \mathrm{Comp}_R(P_R([f]_R), P_R([g]_R), D_{f,g})$. When the pipeline schema fixes the dependency map—as in the canonical pipeline $G \to \Gamma \to \kappa \to E \to H$—the composition operator reduces to a fixed combinator $\otimes_R$.

This compositional provenance structure enables four operational capabilities: *replay equivalence* (identical provenance records operationally witness replay-equivalent execution under deterministic evaluation and collision-resistance assumptions), *divergence localization* (when two executions differ, the first observable divergence is localized to a specific semantic layer), *partial replay* (layers below the divergence point need not be re-executed), and *provenance-preserving registry evolution* (discovery metadata changes preserve all provenance layers). We validate against the empirical measurements of Papers 1–3: 50 replay runs, 1,200 LLM generations, 5 canonical specifications, and registry drift detection across targeted modifications.

All constructions are operational, registry-relative, and replay-grounded. The semantics do not claim universal provenance structures, trustless computation, or cross-system replay guarantees.

1 Introduction

What does it mean to say that two executions have the same provenance?

In existing systems, provenance is a record: a log entry, a run ID, an artifact URI, a DAG of dependencies. Two executions have "the same provenance" if their records match—a comparison that is either tautological (same run ID) or ambiguous (records agree on some fields, disagree on others). When provenance records diverge, the system can report that they differ but not *where* or *why*—the record is flat, and diagnosis requires re-execution.

This paper gives a different answer. Provenance is not a record attached to an execution. It is a *semantic factorization* of execution identity: a decomposition of the content-addressed execution hash into eight semantic layers, each corresponding to a distinct stage of the deterministic pipeline. Two executions have the same provenance if and only if they belong to the same equivalence class in the quotient execution category—the same deterministic execution identity established in Paper 3 [3].

The factorization is compositional. The provenance of a composed pipeline is determined by the provenance of its stages together with the declared dependency map between them. This is not a universal composition law; it is a consequence of the pipeline's typed, staged architecture. When the pipeline schema fixes the dependency map, the composition operator reduces to a fixed combinator. The empirical validation from Papers 1–2—50 replay runs, 1,200 LLM generations, targeted drift detection—becomes interpretable as properties of this compositional provenance structure.

Central thesis. Provenance for deterministic AI execution is a semantic factorization, not execution metadata. The factorization decomposes execution identity into layers that are: (1) compositionally structured (layer provenance composes into pipeline provenance via declared dependencies), (2) replay-grounded (identical provenance operationally witnesses replay-equivalent execution), (3) divergence-localizing (when provenance differs, the first observable difference localizes to a specific semantic layer), and (4) evolution-stable (discovery metadata changes are provenance-invisible).

What this paper is not. This is a provenance semantics and replay semantics paper. It is not a blockchain or distributed-ledger paper—there is no chain, no consensus, no trustless verification. It is not an observability paper—we do not discuss logging, tracing, or monitoring infrastructure. It is not a distributed systems paper—all execution is local and deterministic. The mathematics is restrained: we use maps, composition, and layered hash records, not monads, adjunctions, or enriched categories.

Contributions.

1. A **layered provenance decomposition** P_R mapping execution equivalence classes to 8-layer hash records, where each layer is a semantic dependency boundary (§3).
2. A **dependency-indexed composition law** establishing that pipeline provenance is determined by stage provenance and the declared dependency map (§4).
3. **Replay equivalence** as provenance witness: identical provenance records operationally witness replay-equivalent execution (§5).
4. **Divergence localization**: when two provenance records differ, the first differing layer identifies the semantic boundary where the executions diverge (§6).
5. **Registry evolution semantics** distinguishing provenance-invisible changes (discovery aliases) from provenance-visible changes (canonicalization rules), with a taxonomy of evolution types (§7).
6. **Partial and local replay**: provenance factorization enables re-execution of only the layers that changed (§8).

Notation. We use notation from Paper 3: $\mathcal{B}_R$ is the registry-indexed execution category, $\sim_R$ is the operational equivalence, $\mathcal{B}_R/\sim_R$ is the quotient category, $[f]_R$ denotes the equivalence class of f, κ_R is the canonical representative selection, H is the content-addressed hash, and $\mathcal{E}_R$ is the stochastic proposal space. We suppress the registry subscript R when clear from context. $\mathcal{H}$ denotes the hash space (SHA-256 digests). $\mathcal{H}^8$ denotes 8-tuples of hashes.

2 Replay and Provenance Problems

Paper 2 [2] identified five execution-level problems under stochastic generation (surface-form divergence, provenance fragmentation, morphism explosion, registry-evolution coupling, divergence localization) and solved them with canonicalization, typed specifications, and 8-layer hashing. But those solutions are operational mechanisms; they do not explain the semantic structure that makes them work. This section identifies the provenance-specific problems that the semantic structure must address.

2.1 Logging Is Not Provenance Semantics

Execution logging records what happened: timestamps, parameters, outputs, errors. MLflow [4] stores run artifacts; Weights & Biases [5] stores metrics over time. These systems provide post-hoc observability: a researcher can inspect what was executed. They do not provide:

- **Compositional structure.** A log entry for a pipeline does not decompose into log entries for its stages in a way that supports independent verification. The log is flat: it records the pipeline's inputs and outputs but not the semantic boundaries between stages.

- **Replay semantics.** A log entry asserts that an execution occurred. It does not assert that re-executing with the same inputs will produce the same outputs. Replay requires structural guarantees (determinism, canonicalization, registry stability) that logs do not encode.
- **Divergence localization.** When two log entries differ, the system reports field-level differences. It does not identify which semantic stage the divergence localizes to, or whether stages below the divergence point are replay-equivalent.

The distinction is between *execution traces* (records of what happened, useful for debugging) and *compositional replay semantics* (structured provenance that supports independent layer verification, partial replay, and divergence localization). This paper formalizes the latter.

2.2 The Provenance Decomposition Problem

Paper 2 introduced the 8-layer execution hash as an operational mechanism for fault localization. The mechanism works: when two execution hashes differ, comparing per-layer hashes identifies the divergent layer without re-execution. But the mechanism raises semantic questions it does not answer:

1. *Why eight layers?* The decomposition follows the pipeline architecture, but what semantic property makes this decomposition correct? Could a different decomposition—3 layers, 12 layers—work equally well?
2. *How do layers compose?* If two pipeline stages each have correct provenance, does the composed pipeline automatically have correct provenance? Under what conditions?
3. *What does "same provenance" mean for partial pipelines?* If layers 1–5 match but layers 6–8 differ, are the first five stages replay-equivalent? Can they be skipped in a re-execution?
4. *How does registry evolution interact with provenance?* Which registry changes preserve provenance and which destroy it? Is there a taxonomy?

This paper provides answers by defining provenance as a structured map from execution equivalence classes to layered hash records, with a composition law that relates pipeline provenance to stage provenance.

3 Layered Provenance Decomposition

This section defines the provenance map P_R and establishes its basic properties. The construction is grounded in the 8-layer execution hash from Paper 2 and the quotient execution category from Paper 3.

3.1 The Provenance Map

Definition 3.1 (Provenance map)**.** The *provenance map* is a function

$$P_R : \mathrm{Ob}(\mathcal{B}_R/\!\sim_R) \to \mathcal{H}^8$$

assigning to each execution equivalence class $[f]_R$ an 8-tuple of cryptographic hashes:

$$P_R([f]_R) = (h_1, h_2, h_3, h_4, h_5, h_6, h_7, h_8)$$

where each component is computed from the semantic (SEM), algebraic (ALG), and implementation (IMPL) attributes of the corresponding execution stage. Discovery metadata (DISC) is excluded from all layer computations.

The eight layers correspond to the pipeline stages from Paper 2:

k	Layer hash	Stage	Semantic boundary
1	`DIC_HSH`	Registry	Capability identity: what operations exist and how they are identified
2	`FPR_HSH`	Specification	Execution intent: which family, metric, and parameter ranges
3	`MOR_HSHS`	Morphisms	Execution plan: all expanded, canonicalized pipeline instances
4	`SRH_HSHS`	Search	Search configuration: how the morphism space is explored
5	`CMPN_HSHS`	Composition	Artifact composition: how stage outputs combine
6	`SCR_HSH`	Score	Evaluation: metric computation over execution outputs
7	`SEL_HSH`	Selection	Decision: which candidate is selected as best
8	`DATA_HSH`	Data	Input: content fingerprint of source data

Proposition 3.2 (Well-definedness)**.** *P_R is well-defined on equivalence classes: if $[f]_R = [g]_R$, then $P_R([f]_R) = P_R([g]_R)$.*

Establishment. $[f]_R = [g]_R$ means $f \sim_R g$. By Paper 3's definition of $\sim_R$, equivalent artifacts share SEM/ALG/IMPL attributes (generators E1–E4 preserve these attributes or are defined on attributes that exclude DISC). Since each layer hash is computed from SEM/ALG/IMPL at the corresponding stage, and κ_R produces identical canonical representatives for equivalent artifacts (Paper 3, Definition 5.2), the layer hashes are identical. □

Remark 3.3 (Why eight layers)*.* The number of layers is not arbitrary. Each layer corresponds to a stage in the canonical pipeline $G \to \Gamma \to \kappa \to E \to H$ from Paper 2, expanded to account for the typed intermediate objects (specification, morphisms, search, composition, score, selection) that the pipeline produces. The decomposition follows the pipeline architecture: one layer per semantic boundary where a typed output is produced and content-addressed. A system with a different pipeline architecture would have a different number of provenance layers.

3.2 Provenance as Semantic Factorization

The provenance map factorizes execution identity into independent semantic dimensions. Each layer hash witnesses a specific aspect of the execution:

Definition 3.4 (Layer independence)**.** Two provenance records $P_R([f]_R)$ and $P_R([g]_R)$ *agree at layer k* if their k-th components are equal: $P_R([f]_R)_k = P_R([g]_R)_k$. They *disagree at layer k* if $P_R([f]_R)_k \neq P_R([g]_R)_k$.

The factorization is not merely a decomposition of a hash into substrings. Each layer is computed from a semantically distinct stage of the pipeline. Layer 1 (DIC_HSH) depends on the registry state. Layer 2 (FPR_HSH) depends on the specification and layer 1. Layer 8 (DATA_HSH) depends on the input data. Changing the data changes layer 8 without affecting layers 1–7. Changing the specification changes layer 2 and may cascade to layers 3–7, but leaves layers 1 and 8 unchanged.

This is what distinguishes semantic provenance factorization from execution logging. A log records all fields together. The factorization records them as independent semantic dimensions with declared dependencies.

3.3 Dependency Structure

The layers are not fully independent. They form a dependency structure determined by the pipeline architecture:

Definition 3.5 (Layer dependency)**.** The *layer dependency map* $D_S : \{1, \ldots, 8\} \to \mathcal{P}(\{1, \ldots, 8\})$ specifies, for each layer k, the set of layers whose hashes are inputs to the computation of layer k's hash. In the canonical pipeline:

$D_S(1) = \emptyset$	(registry: no pipeline dependencies)
$D_S(2) = \{1\}$	(specification depends on registry)
$D_S(3) = \{1, 2\}$	(morphisms depend on registry + specification)
$D_S(4) = \{3\}$	(search depends on morphisms)
$D_S(5) = \{4\}$	(composition depends on search)
$D_S(6) = \{5, 8\}$	(score depends on composition + data)
$D_S(7) = \{6\}$	(selection depends on score)
$D_S(8) = \emptyset$	(data: external input, no pipeline dependencies)

The dependency map is *declared*, not inferred. It is a structural property of the pipeline architecture, not a runtime observation. The canonical pipeline's dependency map is fixed: specification always depends on registry, morphisms always depend on specification, and so on. This fixed structure is what makes the provenance composition law tractable.

Remark 3.6 (Layers 1 and 8 are roots)*.* Layers 1 (registry) and 8 (data) have no pipeline dependencies. They are the two external inputs to the execution pipeline. All other layers depend, directly or transitively, on at least one root. This bipartite root structure—one root from the system (registry), one from the environment (data)—is reflected in the divergence localization results: registry changes and data changes produce divergence at different roots, with different cascade patterns.

4 Dependency-Indexed Provenance Composition

This section establishes the central structural result: the provenance of a composed execution is determined by the provenance of its parts together with the declared dependency map between them. This is not a universal composition law. It is a consequence of the typed, staged pipeline architecture and the deterministic hash computation at each stage.

4.1 Composition of Pipeline Stages

In the quotient execution category $\mathcal{B}_R/{\sim_R}$, pipeline stages compose as morphisms: if $[f]_R$ maps specification to morphisms and $[g]_R$ maps morphisms to scores, the composition $[g \circ f]_R$ maps specification to scores. Paper 3 established that composition is well-defined on equivalence classes (the congruence property).

The question for provenance is: how does $P_R([g \circ f]_R)$ relate to $P_R([f]_R)$ and $P_R([g]_R)$?

Definition 4.1 (Dependency-indexed composition)**.** For composable equivalence classes $[f]_R$ and $[g]_R$ with dependency map $D_{f,g} : \{1, \ldots, 8\} \to \mathcal{P}(\{1, \ldots, 8\})$ specifying which output layers of f feed into which input layers of g, the *provenance composition operator* $\mathrm{Comp}_R : \mathcal{H}^8 \times \mathcal{H}^8 \times D \to \mathcal{H}^8$ is defined component-wise:

$$\mathrm{Comp}_R(P_R([f]_R),\, P_R([g]_R),\, D_{f,g})_k \;=\; H\big(P_R([g]_R)_k,\; \{P_R([f]_R)_j : j \in D_{f,g}^{-1}(k)\}\big)$$

where $D_{f,g}^{-1}(k) = \{j : k \in D_{f,g}(j)\}$ is the set of f-layers that feed into g-layer k, and H is the hash combiner (SHA-256 over the concatenation of the layer's own hash and its dependency hashes).

Proposition 4.2 (Dependency-indexed provenance composition)**.** *For composable equivalence classes* $[f]_R$*,* $[g]_R$ *in* $\mathcal{B}_R/{\sim_R}$ *with dependency map* $D_{f,g}$*:*

$$P_R([g \circ f]_R) \;=\; \mathrm{Comp}_R(P_R([f]_R),\, P_R([g]_R),\, D_{f,g})$$

Establishment. Each layer hash of the composed pipeline $[g \circ f]_R$ is computed by the deterministic hash function from: (a) the canonical representative's SEM/ALG/IMPL attributes at that stage, and (b) the

hashes of the stages it depends on. The dependency map $D_{f,g}$ declares exactly which output layers of f serve as inputs to which layers of g. Since hash computation is deterministic and κ_R produces identical canonical representatives for equivalent classes, the composed layer hash is fully determined by the stage's own hash and the dependency hashes from f. This is exactly what Comp_R computes. □

4.2 Fixed-Schema Shorthand

When the pipeline schema S fixes the dependency map—meaning $D_{f,g} = D_S$ for all composable pairs (f, g) conforming to S—the composition operator simplifies:

Definition 4.3 (Pipeline composition shorthand)**.** For pipeline schema S with fixed dependency map D_S:

$$P_R([g]_R) \otimes_R P_R([f]_R) := \text{Comp}_R(P_R([f]_R), P_R([g]_R), D_S)$$

In the canonical BLISP pipeline $G \to \Gamma \to \kappa \to E \to H$, the schema S is the fixed 5-stage cascade from Paper 2, and $\otimes_R$ is the resulting combinator.

Remark 4.4 ($\otimes_R$ is shorthand, not a universal law)*.* $\otimes_R$ is a derived notation for the common case where the dependency map is fixed by the pipeline schema. It is not claimed to form a monoid, group, or category-level structure. It is not claimed to hold for arbitrary pipeline compositions outside the fixed schema. For ad-hoc compositions where the dependency map is not schema-fixed, the full Comp_R form with an explicit dependency map is required.

Example 4.5 (Composition in the canonical pipeline)**.** Consider the composition of specification-to-morphism expansion ($[f]_R$, affecting layers 2–3) with morphism-to-score evaluation ($[g]_R$, affecting layers 4–7). The dependency map $D_{f,g}$ declares that g's layer 4 (search) depends on f's layer 3 (morphisms). The composed provenance at layer 4 is:

$$P_R([g \circ f]_R)_4 = H\big(P_R([g]_R)_4,\ P_R([f]_R)_3\big)$$

The morphism hash from f's output feeds into the search hash of the composed pipeline. If f's morphism layer changes (different parameter grid), g's search layer changes accordingly. If f's morphism layer is unchanged, g's search layer is determined entirely by g's own search configuration.

4.3 Interpretation

The composition law answers a practical question: *can we verify pipeline provenance by verifying stage provenance independently?*

The answer is yes, with the dependency map. If each stage's provenance is verified and the dependency map is known, the pipeline's provenance is determined. This is compositional verification: checking the whole by checking the parts and their declared connections.

Without the dependency map, composition is not well-defined. Two pipeline stages might produce identical provenance records but compose differently because their dependency structure differs. The dependency map is the missing piece that makes provenance composition deterministic.

5 Replay Equivalence

This section connects provenance identity to replay semantics. Provenance is not merely a record to be compared; it is a *replay witness*: under deterministic execution and collision-resistance assumptions, identical provenance operationally witnesses replay-equivalent execution.

5.1 From Provenance to Replay

Definition 5.1 (Replay equivalence)**.** Two executions x_1, x_2 of specification s on data d with registry R are *replay-equivalent* if their provenance records are identical:

$$P_R([x_1]_R) = P_R([x_2]_R)$$

Proposition 5.2 (Provenance as replay witness)**.** *Under deterministic execution and collision-resistance assumptions: if $P_R([x_1]_R) = P_R([x_2]_R)$, then x_1 and x_2 produce identical outputs: $\varepsilon_R(x_1) = \varepsilon_R(x_2)$.*

Establishment. Provenance identity means all eight layer hashes match. Under the collision-resistance assumption (Paper 3, §5.3), matching hashes are treated as operational evidence of identical canonical representatives at every layer. Identical canonical representatives at every layer mean identical inputs to the deterministic evaluation function ε_R. Determinism of ε_R implies identical outputs. □

Remark 5.3 (Provenance comparison replaces re-execution)*.* Replay equivalence is verified by comparing 8-tuples of hashes—a constant-time operation independent of the pipeline's computational cost. This is the operational payoff of semantic provenance: replay verification does not require re-execution. Paper 2 verified this empirically: 50 replay runs (5 specifications × 10 repetitions) produced identical 8-layer provenance records, confirming replay equivalence by hash comparison alone.

5.2 Replay Under Composition

The composition law (Proposition 4.2) extends replay equivalence to composed pipelines:

Corollary 5.4 (Compositional replay)**.** *If two composed pipelines $[g_1 \circ f_1]_R$ and $[g_2 \circ f_2]_R$ have identical stage provenance—$P_R([f_1]_R) = P_R([f_2]_R)$ and $P_R([g_1]_R) = P_R([g_2]_R)$—and the same dependency map D, then they are replay-equivalent:*

$$P_R([g_1 \circ f_1]_R) = P_R([g_2 \circ f_2]_R)$$

Establishment. By Proposition 4.2, both sides equal $\mathrm{Comp}_R(P_R([f]_R), P_R([g]_R), D)$ where $P_R([f]_R) = P_R([f_1]_R) = P_R([f_2]_R)$ and similarly for g. □

This means replay can be verified stage-by-stage. If each stage of a pipeline matches its counterpart, the composed pipeline matches. Independent teams can verify their stages independently; the composition law guarantees that verified stages compose into a verified pipeline.

5.3 Empirical Grounding

Paper 2 established replay determinism empirically across five specifications:

Specification	Parameters	Runs	Unique hashes
MOM_WZS / SRP	3-param sweep	10	1
CARRY / SRP	defaults	10	1
MOM_REV / SRP	3-param sweep	10	1
VOL_TGT / VOL	3-param sweep	10	1
MOM_WZS / MDD	3-param sweep	10	1

In provenance terms: each specification maps to a single provenance record across all repetitions. The fiber $\pi^{-1}([s]_R)$ over each specification's equivalence class contains exactly one provenance record under deterministic replay. This is the operational meaning of the provenance map's well-definedness: identical equivalence classes produce identical provenance.

6 Divergence Localization

When two executions produce different results, the system must identify *where* they diverge. This section formalizes divergence localization as a property of the layered provenance structure.

6.1 Divergence Index

Definition 6.1 (Divergence set)**.** For provenance records $P_R([f]_R)$ and $P_R([g]_R)$, the *divergence set* is:

$$\Delta(f, g) = \{k \in \{1, \ldots, 8\} : P_R([f]_R)_k \neq P_R([g]_R)_k\}$$

the set of layers where the provenance records differ.

Definition 6.2 (First divergence layer)**.** The *first divergence layer* is:

$$k^* = \min \Delta(f, g)$$

the lowest-numbered layer at which the provenance records first observably differ.

Proposition 6.3 (Divergence localization)**.** *If $P_R([f]_R) \neq P_R([g]_R)$, then the first observable provenance divergence is localized to layer*

$$k^* = \min\{k : P_R([f]_R)_k \neq P_R([g]_R)_k\}$$

and the provenance structure localizes the divergence to the execution stage corresponding to layer k^*. Layers $1, \ldots, k^*-1$ are identical. Layers $k^*+1, \ldots, 8$ may or may not diverge depending on whether they depend on layer k^* through the dependency map D_S.

Establishment. Each layer hash is computed from: (a) its stage's canonical SEM/ALG/IMPL attributes, and (b) the hashes of the layers it depends on via D_S. If layers $1, \ldots, k^*-1$ are identical, then layer k^*'s dependency inputs from those layers are identical. A difference at layer k^* with identical dependency inputs indicates that the stage's own canonicalized attributes differ at that layer. Layers above k^* that depend on k^* may cascade; layers that do not depend on k^* are unaffected. □

6.2 Cascade and Independence

The dependency structure determines which layers cascade and which are independent:

Definition 6.4 (Cascade set)**.** The *cascade set* of layer k is:

$$C(k) = \{j \in \{1, \ldots, 8\} : k \in D_S^+(j)\}$$

where D_S^+ is the transitive closure of D_S. $C(k)$ is the set of layers that transitively depend on layer k.

A divergence at layer k^* may cascade to any layer in $C(k^*)$ but cannot affect layers outside $C(k^*) \cup \{k^*\}$. In the canonical pipeline:

Divergent layer k^*	Cascade set $C(k^*)$
1 (registry)	$\{2, 3, 4, 5, 6, 7\}$
2 (specification)	$\{3, 4, 5, 6, 7\}$
3 (morphisms)	$\{4, 5, 6, 7\}$
8 (data)	$\{6, 7\}$

A registry change (layer 1) can cascade through layers 2–7 but cannot affect layer 8 (data). A data change (layer 8) can cascade through layers 6–7 (score, selection) but cannot affect layers 1–5 (registry, specification, morphisms, search, composition).

Example 6.5 (Divergence localization in practice)**.** Paper 2's drift detection experiment introduced targeted changes and compared provenance records. In provenance terms:

Parameter change (different signal window): the first divergence is localized to layer 2 (FPR_HSH). Layers 3–7 cascade (different morphisms, different search, different scores, different selection). Layers 1 (registry) and 8 (data) are unchanged. $k^* = 2$, $\Delta(f, g) = \{2, 3, 4, 5, 6, 7\}$.

Data change (different source file at stable path): the first divergence is localized to layer 8 (DATA_HSH). Layers 6–7 cascade (different scores and selection from different data). Layers 1–5 are unchanged. $k^* = 8$, $\Delta(f, g) = \{6, 7, 8\}$.

Metric change (SRP → MDD): the first divergence is localized to layer 2 (FPR_HSH). Layers 6–7 cascade (different score computation, potentially different selection). Layers 3–5 are unchanged if the morphism grid is metric-independent. $k^* = 2$, $\Delta(f, g) = \{2, 6, 7\}$.

6.3 Diagnostic Value

Divergence localization provides a diagnostic layer: the system identifies *which semantic boundary* changed, not merely *that* the execution differs. The first divergence layer is a pointer into the pipeline architecture:

- $k^* = 1$: the capability registry changed. Check for new operations, removed aliases, or canonicalization rule modifications.
- $k^* = 2$: the specification changed. Check family, metric, or parameter differences.
- $k^* = 3$: the morphism expansion changed. Check parameter grid or canonicalization.
- $k^* = 8$: the data changed. Check source file content.
- $k^* \in \{4, 5, 6, 7\}$: internal pipeline stages changed. These are less common under stable registry and data; they indicate changes in search configuration, composition logic, or scoring implementation.

This diagnostic capability is a direct consequence of the provenance factorization. A monolithic hash provides one bit (match/mismatch). The 8-layer factorization provides at most 3 bits of localization (which of 8 layers diverges first) without re-execution.

7 Registry Evolution Semantics

Registries evolve: aliases are added, descriptions are improved, capabilities are extended. This section formalizes which changes preserve provenance and which destroy it, building on the description/identity separation from Papers 1–2 and the registry monotonicity from Paper 3.

7.1 Two Kinds of Aliases

The corrected semantics from Paper 3 requires a careful distinction between two kinds of aliases that prior work did not separate:

Definition 7.1 (Discovery alias)**.** A *discovery alias* is a mapping in the DISC layer that enables natural-language matching. It is used by the discovery system (AGENT-DISCOVER) to connect user terms to canonical names. Discovery aliases are excluded from capability identity: they do not affect DIC_HSH or any downstream provenance layer.

Example. Adding "sharpe ratio" $\to$ `SRP` as a discovery alias allows the term "sharpe ratio" to discover `SRP`. It does not change what `SRP` computes, does not change its hash, and does not affect any prior execution's provenance.

Definition 7.2 (Canonicalization alias)**.** A *canonicalization alias* is a rewrite rule in the ALG layer that defines expression equivalence. It is used by the normalization pipeline to map surface forms to canonical forms. Canonicalization aliases are part of capability identity: they affect DIC_HSH and may cascade through downstream provenance layers.

Example. Adding the rewrite rule `sr` $\mapsto$ `SRP` as a canonicalization alias means the system now treats `(sr x)` and `(SRP x)` as the same expression. This extends the operational equivalence $\sim_R$ (adding new equivalent pairs), changes the quotient category, and changes DIC_HSH.

Proposition 7.3 (Description invariance)**.** *Let f and f' be execution artifacts differing only in DISC metadata (human descriptions, discovery aliases, UI labels). Then $P_R([f]_R) = P_R([f']_R)$: all eight provenance layers are preserved.*

Establishment. DIC_HSH is computed from SEM $\cup$ ALG $\cup$ IMPL, with DISC excluded. Since f and f' differ only in DISC, DIC_HSH(f) = DIC_HSH(f'). All subsequent layers (2–8) depend directly or transitively on DIC_HSH and their own SEM/ALG/IMPL attributes, which are identical. Therefore all eight layers are preserved. □

Corollary 7.4 (Discovery alias evolution is provenance-invisible)**.** *Adding, removing, or modifying discovery aliases does not change any provenance record. The discovery interface can evolve—improving how capabilities are found by natural-language terms—without invalidating any prior execution's provenance.*

7.2 Registry Evolution Taxonomy

Table 1 classifies registry changes by their provenance effect.

Table 1: Registry evolution taxonomy. Changes are classified by their effect on DIC_HSH (layer 1) and downstream layers. Discovery aliases and descriptions are provenance-invisible because they reside in DISC, which is excluded from identity.

Change type	DIC_HSH	Layers 2–8	Classification
Discovery alias addition	=	=	Provenance-invisible
Description edit	=	=	Provenance-invisible
UI label change	=	=	Provenance-invisible
Canonicalization alias	Δ	May Δ	Layer-1 divergence
Rewrite rule addition	Δ	May Δ	Layer-1 divergence
New capability	Δ	May Δ	Layer-1 divergence
Capability removal	Δ	May Δ	Layer-1 divergence
Implementation variant	Δ	May Δ	Layer-1 divergence
Parameter default change	=	Δ at 2+	Layer-2+ divergence
Data source change	=	Δ at 8	Layer-8 divergence

Remark 7.5 (The grounding gate adds discovery aliases). Paper 1's grounding gate operates through the discovery system, which uses discovery aliases to match user terms to canonical names. Improving the grounding gate's coverage—adding more discovery aliases—is provenance-invisible by Corollary 7.4. This means the admissibility boundary can improve over time (discovering more capabilities from more natural-language terms) without affecting the provenance of any prior execution. The gate evolves independently of execution identity.

7.3 Monotonicity and Merging

Paper 3 established registry monotonicity: registry extension ($R \subseteq R'$) may merge equivalence classes but never splits them. In provenance terms:

Proposition 7.6 (Provenance under registry extension). *Let $R \subseteq R'$ be a registry extension. For equivalence classes $[f]_R$ in $\mathcal{B}_R/\sim_R$:*

- *If the extension adds only DISC-layer changes: $P_R([f]_R) = P_{R'}([f]_{R'})$. All provenance is preserved.*
- *If the extension adds canonicalization aliases or new capabilities: $P_R([f]_R)_1 \neq P_{R'}([f]_{R'})_1$ in general (DIC_HSH changes), and downstream layers may cascade.*
- *Registry extension may cause previously distinct provenance records to become identical (merging: a new alias makes two expressions canonically equivalent). It does not cause previously identical records to diverge (splitting).*

The merging property has a practical implication: adding a canonicalization alias can collapse provenance diversity. If two pipeline expressions previously had distinct canonical forms (and therefore distinct provenance), a new alias might make them canonically equivalent (and therefore provenance-identical). This is a semantic change—the system now treats them as the same execution—and it is reflected in the DIC_HSH change.

8 Partial Replay and Local Replay

The provenance factorization enables two operational capabilities that a monolithic hash does not support: partial replay (re-executing only the layers that changed) and local replay (verifying a subset of layers without the full pipeline).

8.1 Partial Replay

Definition 8.1 (Partial replay)**.** Given provenance records $P_R([f]_R)$ and $P_R([f']_R)$ with first divergence layer $k^* = \min \Delta(f, f')$, a *partial replay* re-executes only layers k^* through 8, reusing the verified provenance of layers 1 through k^*-1.

Partial replay is sound because the provenance factorization declares layer dependencies explicitly. If layers $1, \ldots, k^*-1$ are identical, they produce the same outputs regardless of whether they are re-executed or reused from the prior run. The re-execution starts at the first divergent stage, using the verified outputs of prior stages as inputs.

Example 8.2 (Partial replay under data change)**.** A researcher ran a pipeline last month and recorded its provenance. Today, the source data has been updated (layer 8 changed). Layers 1–5 (registry, specification, morphisms, search, composition) are unchanged. Partial replay re-executes layers 6–7 (score, selection) with the new data, reusing the verified morphism expansion and search configuration. The re-execution cost is proportional to scoring and selection, not to the full pipeline.

Remark 8.3 (Partial replay requires layer outputs, not just hashes)*.* Partial replay requires cached layer *outputs* (the intermediate artifacts), not just their hashes. The provenance hash verifies that the cached output is valid; the cached output provides the actual data for downstream re-execution. A system that stores only hashes can verify replay equivalence but cannot perform partial replay without re-execution from the root.

8.2 Local Replay

Definition 8.4 (Local replay)**.** A *local replay* verifies a contiguous subset of provenance layers k through k' by comparing their hashes against a reference provenance record, without executing or verifying the remaining layers.

Local replay is useful when a researcher wants to verify a specific aspect of an execution without the full pipeline. Examples:

- *Registry verification*: compare layer 1 only. Confirms that the capability set is unchanged since the original execution.
- *Specification verification*: compare layers 1–2. Confirms that both the registry and the execution specification match.
- *Data verification*: compare layer 8 only. Confirms that the source data is unchanged.
- *End-to-end verification*: compare all 8 layers. Confirms full replay equivalence.

Local replay is not the same as partial re-execution. Local replay compares hashes; partial re-execution runs pipeline stages. Local replay answers "are these layers the same?" without any computation. Partial re-execution answers "what do the divergent layers produce with the new inputs?" and requires computation.

8.3 Operational Cost

Operation	Cost	Requires
Full replay verification	$O(1)$ hash comparison	Provenance records only
Local replay (subset)	$O(1)$ hash comparison	Provenance records only
Partial re-execution	Pipeline cost from k^*	Cached layer outputs + new inputs
Full re-execution	Full pipeline cost	Original inputs + registry

The provenance factorization provides a hierarchy of replay operations ordered by cost: local verification (free), full verification (free), partial re-execution (proportional to divergent layers), full re-execution (proportional to entire pipeline). A monolithic hash supports only full verification and full re-execution, with nothing in between.

9 Relationship to Papers 1–3

Papers 1–3 provide the operational and structural foundations on which this paper's provenance semantics rest. Each prior paper established specific mechanisms; this paper reinterprets those mechanisms as properties of compositional provenance.

9.1 Provenance Interpretation of Prior Results

Table 2: Provenance interpretation of empirical and structural results from Papers 1–3.

Prior result	Provenance interpretation	Paper
Grounding gate selectivity (27/96 admitted)	Fiber restriction on provenance: only 27 of 96 provenance classes are reachable through the gated projection	1
Discovery alias additions preserve hashes	Proposition 7.3: DISC changes are provenance-invisible at all 8 layers	1
8-layer execution hash	The provenance map P_R: semantic factorization of execution identity into 8 dependency-bounded layers	2
50-run replay determinism	Proposition 5.2: provenance identity witnesses replay equivalence; all 50 runs produce identical P_R records	2
Drift detection (targeted changes)	Proposition 6.3: divergence localization; parameter changes localize to layer 2, data changes to layer 8	2
278 $\rightarrow$ 235 canonical collapse	Canonicalization merges equivalence classes, reducing the image of P_R from 278 to 235 distinct records	2
1,200 LLM generations at 4 temperatures	Collapse ratio $\gamma = \lvert P\rvert/\lvert\pi(P)\rvert$ measures fiber cardinality: surface diversity over provenance diversity	2
Congruence of $\sim_R$	Foundation for Proposition 4.2: compositional provenance requires that equivalent stages compose into equivalent pipelines	3
Quotient category $\mathcal{B}_R/\sim_R$	Domain of P_R: provenance is defined on equivalence classes, not on raw artifacts	3
Registry monotonicity	Proposition 7.6: registry extension may merge provenance classes but never splits them	3
Description/identity separation (E4)	Proposition 7.3: DISC-only changes preserve all 8 provenance layers	3

9.2 The Provenance Pipeline

Assembling all four papers, the full system decomposes into a provenance-annotated pipeline:

Stage	Paper	Map	Provenance role
Generation	—	G	Outside provenance (stochastic)
Admission	1	Γ	Fiber restriction on provenance classes
Projection	3	π	Maps proposals to provenance classes
Canonicalization	2, 3	κ_R	Selects canonical representative
Factorization	4	P_R	Decomposes identity into 8 layers
Evaluation	2	ε_R	Deterministic execution
Witness	2	H	Content-addressed hash

The provenance map sits between canonicalization and evaluation: it decomposes the canonical execution identity into semantic layers *before* evaluation occurs. This means provenance is available for verification, divergence localization, and partial replay before any execution cost is incurred.

10 Limitations and Non-Claims

Not universal provenance semantics. The provenance map P_R is defined relative to a specific registry R and a specific pipeline architecture (8 stages). Different systems with different architectures would require different provenance decompositions. We do not claim that 8 layers is universal or that Comp_R generalizes to arbitrary pipeline topologies.

Not trustless computation. Provenance verification assumes trust in the execution system: the system's hash computations are correct, the registry is not corrupted, the deterministic pipeline is implemented faithfully. This is a single-system guarantee, not a distributed trust guarantee. We do not use blockchain, consensus, or zero-knowledge proof techniques.

Not cross-system replay. Replay equivalence is defined within a single system with a single registry. Two different systems with different registries, different canonicalization rules, or different hash implementations will generally produce different provenance records for the same specification and data. Cross-system replay would require a shared provenance specification, which this paper does not define.

Not causal inference. Divergence localization identifies the first layer at which provenance observably differs. It does not identify the root cause of the divergence in any causal sense. The statement "the first divergence is localized to layer k^*" means "layer k^* is where the hashes first differ," not "layer k^* caused the difference." The difference between observable localization and causal attribution is significant: a registry change at layer 1 might cascade to layer 6 (score), and the researcher might observe only the layer-6 effect without checking layer 1. The localization identifies layer 1 as the first divergent layer; it does not explain *why* the registry changed.

Not a distributed-ledger. The provenance structure records what was executed within a single deterministic system. It is not a chain, does not accumulate over time in append-only fashion, and does not provide immutability guarantees beyond the collision resistance of SHA-256. Provenance records can be recomputed from inputs; they are not historical artifacts that become irreplaceable.

No universal compositional structure claimed. Comp_R is defined for the specific dependency structure of the canonical pipeline. We do not claim that it forms a monoid, group, semigroup, or any other universal structure. $\otimes_R$ is shorthand for the fixed-schema case and is not claimed to satisfy associativity, commutativity, or other laws beyond what follows from the pipeline's deterministic hash computation. Extending the provenance composition to arbitrary pipeline topologies or non-fixed dependency maps is future work.

Not geometric. The layers are discrete numbered stages, not continuous dimensions. The divergence set $\Delta(f, g)$ is a finite subset of $\{1, \ldots, 8\}$, not a distance metric. We do not introduce topology, continuity, or curvature on the provenance space.

Single domain. The provenance semantics are validated in one domain (systematic trading research) using one pipeline architecture (BLISP). The constructions are domain-independent (provenance maps, composition, divergence localization do not depend on the application domain), but the specific 8-layer decomposition and the empirical validation are BLISP-specific.

Collision resistance is assumed, not proved. The reverse direction of provenance identity (matching hashes imply identical execution) depends on the collision resistance of SHA-256. A hash collision would produce a false replay equivalence—two different executions with identical provenance records. Under standard cryptographic assumptions this is negligible, but it is an assumption, not a theorem.

Partial replay requires infrastructure. Section 8 describes partial replay as a consequence of provenance factorization, but actually performing partial replay requires cached intermediate layer outputs. This paper formalizes the semantic conditions under which partial replay is sound; it does not describe the caching infrastructure needed to implement it.

11 Conclusion

Papers 1–3 established the operational and structural foundations: admissibility boundaries, canonical execution semantics, and quotient execution categories. This paper reveals the provenance structure those systems imply.

The central result is that provenance for deterministic AI execution is a semantic factorization, not execution metadata. The provenance map P_R decomposes execution identity into eight semantic layers with declared dependencies. The provenance of a composed execution is determined by the provenance of its parts together with the declared dependency map: $P_R([g \circ f]_R) = \mathrm{Comp}_R(P_R([f]_R), P_R([g]_R), D_{f,g})$. When the pipeline schema fixes the dependency map, this reduces to the combinator $\otimes_R$.

This compositional structure enables four operational capabilities that distinguish semantic provenance from execution logging:

- *Replay equivalence*: identical provenance operationally witnesses replay-equivalent execution, verified by hash comparison without re-execution.
- *Divergence localization*: the first differing layer identifies the semantic boundary where executions diverge.
- *Partial replay*: only the layers below the first divergence point need re-execution.
- *Provenance-preserving evolution*: discovery metadata changes are invisible at all eight provenance layers, allowing the natural-language interface to improve without invalidating any prior execution's provenance.

The empirical measurements from Papers 1–2—50 replay runs, 1,200 LLM generations, targeted drift detection—are reinterpretable as properties of this provenance structure. The structural results from Paper 3—congruence, quotient categories, registry monotonicity—provide the foundations on which provenance composition rests.

The semantics are deliberately constrained. They are operational (defined by hash computation, not by program behavior), registry-relative (provenance classes change when the registry changes), and replay-grounded (every provenance property maps to a replay operation). They do not claim universal provenance structures, trustless computation, cross-system replay, or causal inference. The provenance factorization formalizes exactly the replay and evolution properties that the deterministic execution system provides—and nothing more.

References

[1] T. Dionysopoulos. The grounding gate: Admissibility and replay guarantees for AI-driven research. *Zenodo*, 2026. doi:10.5281/zenodo.20456984.

[2] T. Dionysopoulos. Canonical execution semantics for stochastic program generators. *Zenodo*, 2026. doi:10.5281/zenodo.20457255.

[3] T. Dionysopoulos. Execution categories for stochastic program generators. *Zenodo*, 2026. doi:10.5281/zenodo.20457403.

[4] M. Zaharia et al. Accelerating the machine learning lifecycle with MLflow. *IEEE Data Engineering Bulletin*, 41(4), 2018.

[5] L. Biewald. Experiment tracking with Weights and Biases, 2020.

[6] E. Dolstra, M. de Jonge, and E. Visser. Nix: A safe and policy-free system for software deployment. In *LISA*, 2004.

[7] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[8] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[9] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *ICDT*, 2001.

[10] L. Moreau and P. Missier. PROV-DM: The PROV data model. W3C Recommendation, 2013.

[11] B. Carpenter et al. Stan: A probabilistic programming language. *Journal of Statistical Software*, 76(1), 2017.

[12] H. Chase. LangChain, 2023.

[13] S. Yao et al. ReAct: Synergizing reasoning and acting in language models. In *ICLR*, 2023.