

Coalgebraic Memory

Part I of *Agentic Engineering*

Matthew Long

The YonedaAI Collaboration, YonedaAI Research Collective

Chicago, IL

`matthew@yonedaai.com`

September 2026

Abstract

Agent memory is often called coalgebraic state, and the description rarely goes past the letters. We fix a functor, a set of laws, and a way of checking both against a system not built for them. A store is a finite partial map from identifiers to records plus a finite set of relation-labelled edges. Two coalgebras sit over that data: a store coalgebra whose alphabet is the memory interface extended by an ambient parameter for allocation, the clock and enumeration order, and a finitary lineage coalgebra whose final semantics is a record's ancestry. Seven laws, eight clauses in all, are settled against **ContextFS** at commit **a93035d**. Three clauses hold. The main one is proved: on ancestry-closed stores the inclusion of the existing identifiers is a homomorphism of ancestry coalgebras, so no later derivation can alter the ancestry of a record, though its descendant structure changes freely. The guarantee covers the four derivation operations and fresh saves, and fails for deletion and for saves under an existing identifier. Five clauses fail, each with a witness in the source. Merge carries payload across a namespace boundary. An undeclared enumeration order settles the merged record's type. Two ancestry representations can disagree, edge endpoints go unchecked, and the stored state is single-axis. For the last failure, two histories differing only in valid time have the same recording. No function of the store can therefore distinguish a retroactive correction from a belated one.

Contents

1	Introduction	3
1.1	Scope	3
1.2	Contributions	3
1.3	Evidence and proof method	4
1.4	Series context	5
2	Background	6
2.1	The memory pillar	6
2.2	Coalgebras	6
2.3	Bitemporal data	7
2.4	Notation	7
3	The formal model	8
3.1	Identifiers, records, relations	8

3.2	The operation alphabet	9
3.3	The store coalgebra	10
3.4	The lineage coalgebra	10
3.5	Bitemporal store observations	11
3.6	The laws	12
3.7	The derivation schema	13
4	The system	14
4.1	Model to code	14
4.2	The relation vocabulary	15
4.3	Derivation	15
4.4	Observation	16
4.5	The agent-facing surface	17
5	Law checking	17
5.1	L1, record monotonicity	18
5.2	Stable ancestry	19
5.3	L2, edge and record agreement	21
5.4	L3, inverse closure	22
5.5	L4, namespace locality	22
5.6	L5, resolution independence	23
5.7	Merge up to bisimilarity	24
5.8	L6, bitemporality	25
5.9	L7, lineage referential integrity	27
5.10	Summary	27
6	Composition with adjacent layers	27
6.1	Layer contribution	27
6.2	Memory composition	28
6.3	Composition with skills	30
6.4	Composition with the harness	31
7	Limits and counterexamples	31
7.1	Bitemporal repair	31
7.2	Repository memory models	32
7.3	Missing run-scoped state	33
7.4	Coalgebraic scope	33
8	Related work	33
9	Open problems	35
10	Conclusion	36
A	Code evidence	36

1 Introduction

An agent that forgets everything between sessions is a function; an agent that remembers is a state-based system, and state-based systems are the subject matter of coalgebra [15, 5]. Banu takes this as the first row of a correspondence between the four externalization pillars of Zhou et al. and the ArchAgents Architecture triple [1, 20, 14]. Memory there is a coalgebra (S, φ) for a polynomial functor, and the correspondence is validated against one reference implementation. It is stated at the level of the pair (S, φ) . It does not fix the functor, the laws, or the means of checking either against a system that was not built to fit the account.

This paper fixes all three for one system. **ContextFS** is a memory layer for coding agents. It stores typed records and exposes them over a command line, a Python facade and a Model Context Protocol server. It also maintains a graph of relation-labelled edges recording how records were derived from one another. Its lineage module offers four derivation operations, **evolve**, **merge**, **split** and **supersede**, each of which creates new records and leaves the originals in place. The coalgebraic account presupposes a persistent carrier, non-destructive derivation and an explicit edge vocabulary. A system with all three is the informative case to examine: where the account fails here, it fails for reasons other than the absence of the structure it needs.

The test isolates a stable derivation fragment rather than a property of the whole system. Derivations add fresh records and ancestry edges, which preserves the ancestry of existing records and confines effects when every named record belongs to one namespace. Deletion and replacement fall outside that fragment: deletion removes incident edges, while a save under an existing identifier replaces a record. Theorem 5.9 proves that the restricted fragment is closed and reachable. The failed clauses identify conditions that the informal state-and-transitions description leaves unstated.

1.1 Scope

Every result below is proved from the definitions in Section 3 or established by the inspection procedure of Section 1.3. Companion work on the other three pillars is cited where a reader may want the adjacent layer.

The claim being tested is narrow. It is not claimed that the four externalization pillars and the Architecture triple (G, Know, Φ) are the same object. Three things are claimed, under stated assumptions. There is a structure-preserving interpretation of the pillars in the triple. A production system realises that interpretation on an identifiable fragment. The places where the interpretation fails are exhibited rather than passed over.

For memory, the identifiable fragment is the lineage fragment: the record store, the edge graph, and the four derivation operations, together with saves that allocate fresh identifiers. Deletion and saves under an existing identifier are outside it. Every positive result below carries that exclusion in its hypotheses (Theorem 5.2, Theorem 5.9). The fragment not covered is retrieval: similarity search over an external vector index is not an observation of the state space in the sense of Theorem 3.14, for the reasons given in Section 7.4. The most-quoted feature of the correspondence is that agent memory is bitemporal in the sense of Jensen and Snodgrass [6]. Its absence here is not merely conceded: valid time is proved to be unrecoverable from the recorded state (Theorem 5.27).

1.2 Contributions

1. A model of a memory store as a pair (m, E) of a finite partial map on identifiers and a finite relation-labelled edge set, with a partial commutative monoid structure given by domain-disjoint union (Theorem 3.4, Theorem 3.6).

2. Two coalgebras over that data and an account of what each is for. The store coalgebra $(S, \alpha: S \rightarrow FS)$ for $FX = (\text{Out} \times X)^{A^+}$ models the API dynamics (Theorem 3.14). The lineage coalgebra (Id, β_s) for $HX = \text{Rec}_\perp \times \mathcal{P}_{\text{fin}}(\mathcal{R} \times X)$ models what can be observed about one memory’s derivation (Theorem 3.18). The alphabet of the first is the memory API extended by an allocation oracle, without which the structure map is not a function (Theorem 3.9).
3. Seven laws that a lineage realisation should satisfy (Theorem 3.21), chosen so that each is decidable by inspection of a candidate implementation rather than by testing.
4. A proof that the ancestry fragment is homomorphism-stable (Theorem 5.7) and that the full lineage fragment is not (Theorem 5.13). This is the positive result of the paper.
5. A separation of the two halves of namespace locality: operations that name only records of one namespace have effects confined to it (Theorem 5.17), while `merge` nevertheless carries payload across the boundary by construction (Theorem 5.18).
6. Three further negative results with code-level witnesses. The merged record’s type is decided by an enumeration order the interface does not declare (Theorem 5.20). Ancestry is recorded in two representations that need not agree (Theorem 5.14). The recorded state is single-axis (Theorem 5.27).
7. An algebraic account of `merge` at the level a coalgebra calls for: it is idempotent up to allocation-blind bisimilarity (Theorem 5.24) and not commutative even up to that equivalence (Theorem 5.25).
8. A frame property for operations that name their identifiers (Theorem 6.2) and two independent failures of it, one for recall by identifier prefix (Theorem 6.3) and one for the lineage observation on stores with dangling edges (Theorem 6.4). Together these are the obstruction to composing memory stores by disjoint union in a multi-tenant setting.
9. An appendix mapping every code-backed claim in the paper to a repository, path, identifier, line and commit.

1.3 Evidence and proof method

The results below are of two kinds and it matters that they are not confused, so we separate them by construction rather than by convention.

Results of the first kind are theorems about the model of Section 3. Their hypotheses are properties of a lineage realisation stated in the model’s own vocabulary, chiefly the *derivation schema* of Theorem 3.23, which says which records and edges each operation adds. They are proved from those hypotheses and from the definitions, with no appeal to any repository. Theorem 5.7, Theorem 5.12, Theorem 5.17, Theorem 5.24, Theorem 7.1 and Theorem 6.2 are of this kind, as are all results in Section 3. They stand or fall on the mathematics.

Results of the second kind are instantiation claims: statements that the source of one repository at one commit exhibits a construct the model names. Their evidence is a path, an identifier and a line range, and their epistemic status is that of a careful reading, not that of a proof. We do not give an operational semantics for Python and SQLite, and we do not claim to have verified anything mechanically. Where such a claim appears in a numbered environment it is because it plays a role in an argument and needs a label. The environment is not a claim of formal derivation. Every instantiation claim in the paper appears as a row of Section A with its file, identifier and lines, so a reader can check each one directly against the repository.

Companion work on the other three pillars is cited in a third way again, and never as a premise. Such a citation reports a fact about a different repository that a reader may want for context, for instance whether one system declares a dependency on another. No definition, hypothesis or proof step below rests on a companion, and every object used here is defined in Section 2 or Section 3.

The two kinds meet in exactly one place. Theorem 5.6 is an instantiation claim: it asserts that `ContextFS` at `a93035d` instantiates the derivation schema. Every positive result about the system in Section 5 is then a theorem of the first kind applied to that lemma. A reader who disputes the reading of the source can reject Theorem 5.6 without any of the mathematics changing, and a reader who accepts it gets the theorems. Negative results are simpler: each exhibits a concrete counterexample whose only code-dependent ingredient is a single named expression.

1.4 Series context

This paper is the first of four on the externalization pillars, and the papers share notation. The results below do not depend on formal claims from the companion papers.

Nothing formal is imported. The proofs need **Set**, the definition of a coalgebra and its homomorphisms, and the bitemporal vocabulary of the temporal database literature, all recalled in Section 2.

The companion treatment of skills as operations of a coloured operad [10] uses the memory carrier as one colour. The type **Mem** of a handle on the carrier is therefore defined here (Theorem 3.15), as a single colour rather than a family indexed by record schemas. Port types themselves are defined in the companion treatment of protocols [11], together with the alphabet **T**. The notion is used here in that sense, and the symbol **T** is not reused for anything else.

The companion treatment of the harness [12] develops the Architecture triple (G, Know, Φ) in full, together with the categories **Arch** and **Sys** and the reading of an agent as a lax monoidal functor $A: \mathbf{Arch} \rightarrow \mathbf{Sys}$. None of that is needed here and none of it is used. One boundary is needed, and we fix it ourselves so that nothing below depends on a companion. A *certificate* *cert* in the sense of Banu’s Definition 1 is a checkable statement about an architecture, consisting of a claim, an assignment of its parameters, and evidence that can be replayed [1]. The *Know* component of an architecture is a collection of such statements. The state space S of this paper is a different thing: it is what is remembered, not what is checked. In `ContextFS`, the schema registry and the structured-data validation are *Know*-level and lie outside S . The record store and the edge set are S .

Theorem 5.27 is proved here rather than assumed anywhere, and is what companion work cites when it describes the memory component of a harness as a single-axis store.

What this layer offers a composite is the carrier together with one stability guarantee. A skill that reads the ancestry of a record at one point in a composite and again later is entitled, by Theorem 5.7, to assume the ancestry has not changed. It is not entitled to assume the same of descendants, and Theorem 6.3 records what identifier discipline is needed for even the first assumption to be meaningful when several tenants share a store.

Throughout, statements about code are claims about the source at the pinned commit. We write “the implementation at `a93035d` does X ” and mean that a reader of the named file and line will find X written there. We do not report runtime observations, and we do not describe any property as established by anything stronger than a reading of the source.

2 Background

2.1 The memory pillar

Zhou et al. survey the externalization of agent capability into four components that sit around a language model rather than inside it: memory, skills, protocols, and the harness that composes them [20]. Memory in that taxonomy is whatever survives the boundary of a single model call. It is not the context window, which is an input. It is the store the agent consults to build the input and updates after acting.

Two things follow from that definition. Memory is stateful in a strong sense: the same query at two times may legitimately return different answers, and the difference is information rather than error. Memory is also observed rather than inspected. An agent does not read the store. It calls an operation and receives an answer. Both properties are the defining properties of the systems coalgebra was designed for [15, 5], which is why the identification of memory with coalgebraic state is natural rather than decorative.

Banu takes the identification as the first row of a four-row correspondence between Zhou’s pillars and the ArchAgents Architecture triple (G, Know, Φ) of de los Riscos, Corbacho and Arbib [1, 14]. In that correspondence memory is modelled as a coalgebra for a polynomial functor, a pair (S, φ) with $\varphi: S \rightarrow \mathcal{P}(S)$. The reference implementation is said to carry bitemporal records, so that the question “what did the agent believe at time t ” has an answer. The correspondence is stated at that level of detail and validated against a single reference implementation, which the author identifies as a limitation of the work [1, Section 7.2]. The identifiers named there, including `BiTemporalMemory` and `RunContext`, belong to that reference implementation. They do not occur in `ContextFS` at `a93035d`, and this paper does not use them (Theorem 7.3).

Cao argues the wider motivation for externalizing memory on complexity grounds [2]. The argument contrasts systems whose decision logic is fixed in source with systems that generate it at run time, and treats the growth of interaction paths against roughly constant human capacity as the reason the second kind displaces the first. We take that as context, not as a premise. Nothing below depends on it.

2.2 Coalgebras

We use only the elementary theory, in the form set out by Rutten [15] and Jacobs [5], and record it here to fix notation rather than to introduce it. Fix the category **Set**. For a functor $F: \mathbf{Set} \rightarrow \mathbf{Set}$, an F -coalgebra is a pair (X, c) with $c: X \rightarrow FX$, and a homomorphism $f: (X, c) \rightarrow (Y, d)$ is a function with $d \circ f = Ff \circ c$. These form a category $\mathbf{Coalg}(F)$. A final F -coalgebra is a terminal object $(\nu F, \iota)$ of that category. When it exists, each (X, c) carries a unique homomorphism $\llbracket - \rrbracket_c: X \rightarrow \nu F$, the behaviour map. An F -bisimulation between (X, c) and (Y, d) is a relation $\mathcal{B} \subseteq X \times Y$ carrying an F -coalgebra structure making both projections homomorphisms.

Two standard facts are used. First, every finitary **Set**-endofunctor, that is, one preserving filtered colimits, is accessible and therefore has a final coalgebra. Second, when F preserves weak pullbacks, bisimilarity coincides with behavioural equivalence, so two states have equal behaviour exactly when some bisimulation relates them. Both are in [15, 5].

The homomorphism condition carries the weight below. If c says what can be observed at a state and where the state can go, then $d \circ f = Ff \circ c$ says that observing at $f(x)$ agrees with observing at x . It also says that the successors of $f(x)$ are the f -images of the successors of x . Several results below are failures of exactly this condition for maps one would like to be homomorphisms, such as “restrict the store to one namespace”.

2.3 Bitemporal data

The temporal database literature distinguishes two independent time dimensions [6, 17]. *Valid time* is when a fact is asserted to hold in the modelled world. *Transaction time* is when the system recorded the assertion. A relation is *bitemporal* when each tuple carries timestamps in exactly one valid-time and one transaction-time dimension. The distinction is not decoration. It is what separates two situations that a single-axis log conflates: a fact about the past that we learn late, and a fact about the present that has just changed.

For an agent the distinction has an operational reading. Transaction time answers “what did the system have on file at t ”. Valid time answers “what was actually the case at t ”. Their combination answers “what did the system believe at t_2 about the state of the world at t_1 ”, which is the question an audit of an agent’s decision requires. We use the series symbols τ_v, τ_t for the two axes.

Definition 2.1 (Bitemporal record). Let $\mathbb{T}$ be a linear order of time instants and Val a set of values. A *bitemporal record* over Val is a triple $(v, \tau_v, \tau_t) \in \text{Val} \times \mathbb{T} \times \mathbb{T}$ read as: the value v is asserted to hold from valid time τ_v , and that assertion was recorded at transaction time τ_t . A record schema is *single-axis* when it carries no field whose value is under the caller’s control and is read as a valid time.

Definition 2.2 (Bitemporal history and its snapshot). A *bitemporal history* is a finite sequence $h = ((v_1, p_1, t_1), \dots, (v_n, p_n, t_n))$ of bitemporal records with $t_1 < \dots < t_n$. For $(\tau_v, \tau_t) \in \mathbb{T} \times \mathbb{T}$ put

$$J_h(\tau_v, \tau_t) = \{j : t_j \leq \tau_t \text{ and } p_j \leq \tau_v\},$$

the assertions recorded by τ_t whose validity had begun by τ_v . The *snapshot function* $\sigma_h : \mathbb{T} \times \mathbb{T} \rightarrow \text{Val}_\perp$ is $\perp$ when $J_h(\tau_v, \tau_t)$ is empty, and otherwise v_k where

$$P = \max\{p_j : j \in J_h(\tau_v, \tau_t)\}, \quad k = \max\{j \in J_h(\tau_v, \tau_t) : p_j = P\}.$$

The two maxima are taken in the stated order and neither may be omitted. The inner one selects on the valid-time axis: among the assertions in force at τ_v , the governing one is the one whose validity began latest, since a later assertion about τ_v supersedes an earlier one about the same instant. The outer one selects on the transaction-time axis and breaks the remaining tie: among assertions with the same valid-time start, the one recorded last is the current belief. Maximising over j alone would conflate the axes. It would let a correction recorded late about a distant past instant shadow an assertion about an instant nearer to τ_v , which is the confusion the two-axis model exists to prevent.

Remark 2.3. Theorem 2.2 uses valid-time instants rather than intervals. This is the weaker setting, and it makes the negative result of Theorem 5.27 stronger. A system that cannot record the instant from which an assertion is claimed to hold certainly cannot record an interval.

2.4 Notation

$\mathbb{T}$ is a linear order. For a set X we write $X_\perp = X + \{\perp\}$, X^* for the set of finite sequences over X , and $\mathcal{P}_{\text{fin}}(X)$ for the set of finite subsets of X . For sets X, Y we write $X \rightarrow_{\text{fin}} Y$ for the set of finite partial functions and $\text{dom}(f)$ for the domain of f . Repository names are set as **ContextFS**; code identifiers and paths as `memory_lineage.py`. All ContextFS line references are to commit `a93035d`.

3 The formal model

3.1 Identifiers, records, relations

Definition 3.1 (Basic data). Fix a countably infinite set Id of *identifiers*, a set Rec of *records*, and a finite set $\mathcal{R}$ of *relation labels*. Records carry at least the fields

$$\text{id}: \text{Rec} \rightarrow \text{Id}, \quad \text{ns}: \text{Rec} \rightarrow N, \quad \text{ty}: \text{Rec} \rightarrow T, \quad \text{ct}: \text{Rec} \rightarrow \mathbb{T},$$

an identifier, a namespace drawn from a set N , a type drawn from a finite set T , and a creation instant. Records carry further fields (content, tags, summary, a metadata map) that the model treats as opaque payload.

Definition 3.2 (Inverse structure on labels). An *inverse structure* on $\mathcal{R}$ is a function $\overline{(-)}: \mathcal{R} \rightarrow \mathcal{R}$ with $\overline{\overline{\rho}} = \rho$ for all ρ , that is, an involution. Labels with $\overline{\rho} = \rho$ are *symmetric*.

Definition 3.3 (Ancestry labels). An *ancestry selection* is a subset $\mathcal{R}^- \subseteq \mathcal{R}$ closed under no operation in particular, singled out as the labels whose direction points from a derived record to the records it was derived from. We write $\mathcal{R}^+ = \{\overline{\rho} : \rho \in \mathcal{R}^-\}$ for the corresponding descendant labels and assume $\mathcal{R}^- \cap \mathcal{R}^+ = \emptyset$.

Definition 3.4 (Store). A *store* is a pair $s = (m_s, E_s)$ with $m_s \in \text{Id} \rightarrow_{\text{fin}} \text{Rec}$ satisfying $\text{id}(m_s(i)) = i$ for all $i \in \text{dom}(m_s)$, and $E_s \in \mathcal{P}_{\text{fin}}(\text{Id} \times \mathcal{R} \times \text{Id})$. We write S for the set of stores. A store is *ancestry-closed* when for every $i \in \text{dom}(m_s)$ and every $(i, \rho, j) \in E_s$ with $\rho \in \mathcal{R}^-$ we have $j \in \text{dom}(m_s)$. A store is *inverse-closed* when $(i, \rho, j) \in E_s$ implies $(j, \overline{\rho}, i) \in E_s$.

Neither closure condition is imposed on S . Both are properties that a realisation may or may not maintain, and Section 5 reports which of them **ContextFS** maintains.

Definition 3.5 (Domain-disjoint union). For stores s, s' with $\text{dom}(m_s) \cap \text{dom}(m_{s'}) = \emptyset$, set $s \uplus s' = (m_s \cup m_{s'}, E_s \cup E_{s'})$. Otherwise $s \uplus s'$ is undefined.

We define $\uplus$ with this side condition because disjointness of record identifiers is what a reading of **ContextFS**'s own tenancy boundary suggests, and because seeing precisely where it is too weak is one of the results of Section 6. It is not the operator we end up recommending. Theorem 6.4 shows that $\uplus$ does not preserve lineage behaviour, and Theorem 6.5 refines it to the operator that does. The refinement requires disjointness of the identifiers a store mentions in either component rather than only in its record map. A reader who wants the compositional structure alone, without the diagnosis, may take Theorem 6.5 as the definition and Theorem 6.6 as its property.

Proposition 3.6 (Stores form a partial commutative monoid). $(S, \uplus, \emptyset)$ is a partial commutative monoid: $\uplus$ is commutative and associative where defined, and the empty store $\emptyset = (\emptyset, \emptyset)$ is a two-sided unit.

Proof. Commutativity and the unit law are immediate from the definition, since union of partial functions with disjoint domains and union of finite sets are commutative with the empty map and empty set as units. For associativity, both $(s \uplus s') \uplus s''$ and $s \uplus (s' \uplus s'')$ are defined exactly when $\text{dom}(m_s)$, $\text{dom}(m_{s'})$ and $\text{dom}(m_{s''})$ are pairwise disjoint, and in that case both equal $(m_s \cup m_{s'} \cup m_{s''}, E_s \cup E_{s'} \cup E_{s''})$. $\square$

$\uplus$ places no condition on edges. Two stores with disjoint record domains may carry edges pointing into each other's records, and their union will too. This is deliberate: the implementation places no such condition either (Theorem 5.30).

3.2 The operation alphabet

Definition 3.7 (Lineage signature). The *lineage signature* Σ consists of the following operation symbols with the indicated argument sorts, where C is a set of contents, Θ a set of merge strategies, Ξ a set of change reasons and W a set of edge weights:

$\text{save}(c, \theta_r)$	$c \in C, \theta_r \in \text{Rec-attributes},$
$\text{recall}(i)$	$i \in \text{Id},$
$\text{evolve}(i, c, \xi)$	$i \in \text{Id}, c \in C, \xi \in \Xi,$
$\text{merge}(\vec{i}, c, \theta)$	$\vec{i} \in \text{Id}^*, c \in C_\perp, \theta \in \Theta,$
$\text{split}(i, \vec{c})$	$i \in \text{Id}, \vec{c} \in C^*,$
$\text{supersede}(i, j, r)$	$i, j \in \text{Id}, r \in C_\perp,$
$\text{link}(i, j, \rho, w, b)$	$i, j \in \text{Id}, \rho \in \mathcal{R}, w \in W, b \in \{0, 1\}.$

We write A for the set of fully applied calls of these symbols.

Definition 3.8 (Ambient parameter). An *ambient parameter* is a triple $\omega = (\vec{k}, t, \pi)$ consisting of a finite sequence $\vec{k} \in \text{Id}^*$ of identifiers to be allocated, an instant $t \in \mathbb{T}$ to be used as the clock reading, and a *resolution parameter* $\pi \in \Pi$. Here Π is a set of enumeration data: an element of Π fixes, for every finite set of values the implementation may enumerate, a linear order on that set. We call $(\vec{k}, t)$ the *allocation part* of ω and π its *resolution part*. Write Ω for the set of ambient parameters and $A^+ = A \times \Omega$ for the *extended alphabet*.

Remark 3.9 (Why the ambient parameter is necessary, and where to draw its boundary). Without Ω there is no function $S \rightarrow (\text{Out} \times S)^A$ to be had. Each derivation operation constructs records whose identifiers come from a fresh-name generator and whose timestamps come from the system clock, so the successor state is not determined by the store and the call. One can respond by moving to a nominal or named-set semantics, by quotienting states by identifier renaming, or by making the ambient data explicit. We make it explicit because it is the option that keeps the model in **Set** and keeps the laws below decidable by inspection.

The choice of what Ω absorbs is not free, and drawing it too narrowly would be a way of smuggling a negative result into a definition. If any source of ambient variation is left outside Ω then α is not a function and the coalgebraic reading fails before any law is evaluated. We therefore let Ω absorb every source we found: allocation, the clock, and the enumeration orders that the implementation consults when it iterates a finite collection whose iteration order the language does not fix. The resolution component π is what makes α total and single-valued in the presence of the two constructs identified in Theorem 5.20 and Theorem 6.3.

Absorbing a source of variation into the alphabet is not the same as excusing it. The substantive question is which components of ω a well-behaved realisation is allowed to read, and that is what L5 (Theorem 3.21) asks. The allocation part is data the interface declares, since records carry identifier and timestamp fields. The resolution part is not data the interface mentions at all. A realisation that reads π has a parameter its own specification does not admit to having.

Remark 3.10 (Three alternative presentations). Three other presentations are available, each with a different cost.

The first is to keep the alphabet at A and observe that α is then not a function, so the system is not a coalgebra and there is nothing further to say. That conclusion is correct and uninformative. It applies equally to every system that allocates identifiers. It also cannot distinguish the part of **ContextFS** that satisfies Theorem 5.7 from the expression in **merge** that does not satisfy L5, because it refuses both at the same point. A model whose only verdict is a single global refusal is not a model of anything.

The second is to keep the alphabet at A and move the variation into the codomain, taking a coalgebra $S \rightarrow \mathcal{M}(\text{Out} \times S)^A$ for a suitable monad $\mathcal{M}$. For the variation at issue this is not an alternative but the same thing written differently. Currying gives a bijection between functions $S \rightarrow (\text{Out} \times S)^{A \times \Omega}$ and functions $S \rightarrow ((\text{Out} \times S)^\Omega)^A$, and $(-)^\Omega$ is the reader monad on Ω . The presentation of Theorem 3.14 is the uncurried form of a coalgebra valued in that monad. We work uncurried because the laws are then equations between functions of a named parameter, which is what makes “the result depends on this component of ω and not that one” a statable property rather than an informal gloss.

A third presentation combines the two, using the reader monad for the allocation part and the finite powerset for the rest, giving $S \rightarrow \mathcal{P}_{\text{fin}}(\text{Out} \times S)^{A \times \text{Id}^* \times \mathbb{T}}$. This is a perfectly serviceable alternative and L5 remains statable in it, as the requirement that the value be a singleton for every argument. We do not claim our formulation is the only one available. We prefer it for one reason. The powerset records that two outcomes are possible and forgets what selects between them, so a realisation that reads an enumeration order and one that flips a coin receive the same description. But Theorem 5.20 and Theorem 6.3 are about a specific ambient datum being read, and it is convenient to be able to name it.

Definition 3.11 (Outputs). Let $\text{Out} = \text{Rec} + \text{Rec}^* + (\text{Id} \times \mathcal{R} \times \text{Id}) + \{\text{ack}\} + \{\text{err}\}$, the disjoint union of a record, a finite list of records, an edge, an acknowledgement and a single error value err . The two derived-observation cases $\text{Rec}_\perp$ used by `recall` embed in Out via $\text{Rec} + \{\text{err}\}$.

3.3 The store coalgebra

Definition 3.12 (Behaviour functor). Define $F: \mathbf{Set} \rightarrow \mathbf{Set}$ by $FX = (\text{Out} \times X)^{A^+}$ on objects and, for $f: X \rightarrow Y$, by $Ff(g) = (\text{id}_{\text{Out}} \times f) \circ g$.

Lemma 3.13. F is a functor.

Proof. F is the composite $(-)^{A^+} \circ (\text{Out} \times -)$ of the product functor with a fixed set and the exponential functor with a fixed exponent, both of which are functors on $\mathbf{Set}$; the stated action on morphisms is the composite action. Explicitly, $F\text{id}_X(g) = (\text{id} \times \text{id}) \circ g = g$ and $F(f' \circ f)(g) = (\text{id} \times f' \circ f) \circ g = (\text{id} \times f') \circ (\text{id} \times f) \circ g = Ff'(Ff(g))$. $\square$

Definition 3.14 (Store coalgebra). A *lineage realisation* is an F -coalgebra (S, α) on the set of stores. We write $\alpha(s)(a, \omega) = (\text{out}(s, a, \omega), \text{nx}(s, a, \omega))$ for the two components and call out the *output map* and nx the *transition map*. The *memory coalgebra* of a system is its lineage realisation.

Definition 3.15 (Memory port type). **Mem** is the type of a handle on the carrier S of a memory coalgebra: a value of type **Mem** determines a store and admits the calls of Theorem 3.7. **Mem** is exported as a single type, not a family indexed by record schemas.

Theorem 3.15 is the only thing later Parts need from this Part in order to speak about memory as a component. It deliberately says nothing about record schemas: schema validation belongs to the Know component in the sense fixed by Part IV, not to the state space.

3.4 The lineage coalgebra

The store coalgebra models what the API does. It does not directly model what a lineage query returns, because a lineage query is about one memory, not about the whole store. We therefore give a second coalgebra, on the identifiers of a fixed store.

Definition 3.16 (Lineage functor). Define $H: \mathbf{Set} \rightarrow \mathbf{Set}$ by $HX = \text{Rec}_\perp \times \mathcal{P}_{\text{fin}}(\mathcal{R} \times X)$, with $Hf = \text{id}_{\text{Rec}_\perp} \times \mathcal{P}_{\text{fin}}(\text{id}_{\mathcal{R}} \times f)$. For $\mathcal{R}^- \subseteq \mathcal{R}$ define the *ancestry functor* $H^-X = \text{Rec}_\perp \times \mathcal{P}_{\text{fin}}(\mathcal{R}^- \times X)$ likewise.

Lemma 3.17. H and H^- are finitary $\mathbf{Set}$ -endofunctors preserving weak pullbacks. Consequently each has a final coalgebra, and for each of them bisimilarity coincides with behavioural equivalence.

Proof. $\mathcal{P}_{\text{fin}}$ is finitary and preserves weak pullbacks; constant functors and finite products are finitary and preserve weak pullbacks; both properties are closed under composition and finite product. Hence H and H^- have both properties. A finitary $\mathbf{Set}$ -endofunctor is accessible, hence has a final coalgebra, and weak-pullback preservation gives the coincidence of bisimilarity with behavioural equivalence [15, 5]. $\square$

Definition 3.18 (Lineage coalgebra of a store). For a store s define $\beta_s: \text{Id} \rightarrow H\text{Id}$ by

$$\beta_s(i) = (m_s(i), \{(\rho, j) : (i, \rho, j) \in E_s\}),$$

where $m_s(i) = \perp$ when $i \notin \text{dom}(m_s)$. The *ancestry coalgebra* is $\beta_s^-: \text{Id} \rightarrow H^-\text{Id}$, defined the same way but with the second component restricted to labels in $\mathcal{R}^-$. We write $\llbracket i \rrbracket_s \in \nu H$ and $\llbracket i \rrbracket_s^- \in \nu H^-$ for the corresponding behaviours.

Concretely, $\llbracket i \rrbracket_s^-$ is the ancestry unfolding of i : the record at i , together with the labelled ancestry unfoldings of everything i was derived from, to any depth. This is the object a lineage query approximates, and it is the object Theorem 5.7 says is stable.

Remark 3.19 (The cost of totality). β_s is defined on all of Id , including identifiers with no record, in which case the first component is $\perp$. This makes β_s total without requiring stores to be ancestry-closed, which is necessary because **ContextFS** does not enforce referential integrity on edges (Theorem 5.30).

The choice has a consequence that we use rather than ignore. If two distinct absent identifiers j, j' have bisimilar outgoing edge structure, which happens in particular when neither occurs as the source of an edge of E_s , then $\beta_s(j)$ and $\beta_s(j')$ are bisimilar and behaviour identifies them. Absence alone does not force this: Theorem 3.4 places no referential constraint on E_s , so an identifier outside $\text{dom}(m_s)$ may perfectly well be the source of edges and then carries a nontrivial behaviour below a $\perp$ observation. What the choice does force is that the *observation* at an absent identifier is the same $\perp$ whichever identifier it is, so behaviour never records which record is missing, only that one is. One could avoid the identification by taking $HX = (\text{Rec} + \text{Id}) \times \mathcal{P}_{\text{fin}}(\mathcal{R} \times X)$, recording which identifier is missing. The behaviour would then distinguish states that differ only in the name of an absent record. That defeats the purpose of an allocation-blind observation (Theorem 5.23) and would make Theorem 5.24 false.

Where the identification could distort a result, we exclude it by hypothesis. Every statement in Section 5.2 assumes an ancestry-closed store, and on such a store no ancestry behaviour contains a $\perp$ observation at all, so the question does not arise. The one place it does real work is Theorem 6.4. There it is the $\perp$ observation and not the successor structure that carries the argument: a $\perp$ first component and a record first component differ whatever the outgoing edges are. The bisimulation exhibited in Theorem 5.24 pairs each absent identifier with itself, so no two distinct absent identifiers are ever identified there.

3.5 Bitemporal store observations

Definition 3.20 (Bitemporal observation). Let (S, α) be a lineage realisation and $L \subseteq \text{Id}$ a set of lineage roots. A *bitemporal observation* for (S, α) is a map $\text{obs}: S \times L \times \mathbb{T} \times \mathbb{T} \rightarrow \text{Rec}_\perp$, read:

$\text{obs}(s, \ell, \tau_v, \tau_t)$ is the record that, according to s , was believed at transaction time τ_t to describe the state of affairs at valid time τ_v for the lineage rooted at ℓ . The observation is *degenerate in valid time* when $\text{obs}(s, \ell, \tau_v, \tau_t) = \text{obs}(s, \ell, \tau'_v, \tau_t)$ for all τ_v, τ'_v . Given a map Ψ that records histories under lineage roots, the observation is *sound for Ψ* when $\text{obs}(\Psi(h), \ell, \tau_v, \tau_t) = \sigma_h(\tau_v, \tau_t)$ for every recorded history h , its root ℓ , and both time arguments.

3.6 The laws

We now fix what it means for a lineage realisation to be well behaved. The laws are chosen so that each can be settled for a candidate implementation by reading its source, and so that each failure has an operational consequence rather than only an aesthetic one.

Definition 3.21 (Laws for a lineage realisation). Let (S, α) be a lineage realisation with output map out and transition map nx . Let $D \subseteq A$ be the *derivation calls*, those of the symbols *evolve*, *merge*, *split* and *supersede*. The realisation satisfies:

L1 *Record monotonicity*. For every s , every $a \in D$ and every ω whose allocated identifiers lie outside $\text{supp}(s)$, writing $t = \text{nx}(s, a, \omega)$: $\text{dom}(m_s) \subseteq \text{dom}(m_t)$ and $m_t(i) = m_s(i)$ for every $i \in \text{dom}(m_s)$. No derivation call overwrites or deletes a record.

L2 *Edge and record agreement*. For every reachable s and every $i \in \text{dom}(m_s)$: the ancestry recorded in the payload of $m_s(i)$ and the ancestry recorded in E_s agree. That is, j is named as a derivation source in the payload of $m_s(i)$ if and only if $(i, \rho, j) \in E_s$ for the corresponding $\rho \in \mathcal{R}^-$.

L3 *Inverse closure*. Every store reachable from the empty store by derivation calls whose edge writes complete is inverse-closed in the sense of Theorem 3.4.

L4 *Namespace locality*. Two clauses. (i) *Confinement*. For every $\nu \in N$, writing $A_\nu(s) \subseteq A$ for the calls all of whose identifier arguments name records of s in the fibre $\text{ns}^{-1}(\nu)$, the square

$$\begin{array}{ccc} S & \xrightarrow{\alpha(-)(a, \omega)} & \text{Out} \times S \\ \downarrow \pi_\nu & & \downarrow \text{id} \times \pi_\nu \\ S & \xrightarrow{\alpha(-)(a, \omega)} & \text{Out} \times S \end{array}$$

commutes at s for every $a \in D \cap A_\nu(s)$ and every ω that allocates outside $\text{supp}(s)$ and resolves identifiers by equality. Thus π_ν is a homomorphism for the derivation fragment that names only ν -records. (ii) *Construction locality*. For every call a and every record r constructed by $\alpha(s)(a, \omega)$, the payload of r is determined by the records of s lying in the fibre $\text{ns}^{-1}(\text{ns}(r))$, by the non-identifier arguments of a , and by ω .

L5 *Resolution independence*. For all s , a and all ambient parameters $\omega = (\vec{k}, t, \pi)$ and $\omega' = (\vec{k}, t, \pi')$ with the same allocation part, $\alpha(s)(a, \omega) = \alpha(s)(a, \omega')$. A realisation may read the identifiers it is given and the clock; it may read no other ambient datum.

L6 *Bitemporality*. There is a recording map Ψ and a bitemporal observation for (S, α) that is sound for Ψ , not degenerate in valid time, and distinguishes a retroactive correction from a belated recording of a subsequent fact.

L7 *Lineage referential integrity*. Every reachable store is ancestry-closed in the sense of Theorem 3.4.

Definition 3.22 (Namespace restriction). For $\nu \in N$ define $\pi_\nu(s) = (m_s \upharpoonright_\nu, E_s \upharpoonright_\nu)$ where $m_s \upharpoonright_\nu$ is the restriction of m_s to $\{i \in \text{dom}(m_s) : \text{ns}(m_s(i)) = \nu\}$ and $E_s \upharpoonright_\nu$ is the restriction of E_s to triples both of whose endpoints lie in $\text{dom}(m_s \upharpoonright_\nu)$.

L1 and L7 are structural: they say the carrier grows and stays connected. L2 and L3 are redundancy laws: they say that two ways of recording the same fact do not diverge. L4 and L5 are locality laws: they say an operation depends only on what it names and on what its interface admits to reading. L6 is the temporal law.

Two remarks on the statement of L4. Its first clause is restricted to $A_\nu(s)$ deliberately. Asking π_ν to be an endomorphism for the whole alphabet would make the law false for any realisation whatever, since $\text{recall}(i)$ for i outside the fibre returns a record before restriction and an error after it. A law that no realisation can satisfy tells one nothing about the realisation at hand. The interesting question is whether operations that stay inside a namespace have effects that stay inside it, and that is clause (i). Clause (ii) is the separate question of whether an operation can pull payload across the boundary, and it is the clause that fails below. The seven laws are not independent in a logical sense, but no one of them entails another for the implementation we examine, and each is settled separately in Section 5.

3.7 The derivation schema

The positive results of Section 5 are theorems about any realisation whose derivation operations have the following shape. Isolating that shape as a hypothesis lets the mathematics be proved once, independently of any repository, and lets the reading of the source enter at exactly one point (Theorem 5.6).

Definition 3.23 (Derivation schema). A lineage realisation *follows the derivation schema* when, for every store s , every derivation call $a \in D$ and every ambient parameter ω whose allocated identifiers lie outside $\text{supp}(s)$, writing $t = \text{nx}(s, a, \omega)$ and $I(a) \subseteq \text{Id}$ for the identifiers named in a :

- S1** *Guarded retrieval.* No record outside $I(a) \cap \text{dom}(m_s)$ is read. If the call constructs a record and some $i \in I(a)$ is absent from $\text{dom}(m_s)$, then either the output is `err` and $t = s$, or the absent identifier is dropped and the call proceeds on the remainder. A call that constructs no record need not retrieve anything.
- S2** *Fresh construction.* m_t extends m_s , and every identifier in $\text{dom}(m_t) \setminus \text{dom}(m_s)$ is one of the identifiers allocated by ω . Each constructed record is a function of the retrieved records, the non-identifier arguments of a and ω .
- S3** *Namespace inheritance.* Each constructed record has the namespace of one of the retrieved records.
- S4** *Edge polarity.* $E_s \subseteq E_t$, and every triple in $E_t \setminus E_s$ is either (κ, ρ, i) with $\rho \in \mathcal{R}^-$, κ allocated by ω and $i \in I(a) \cap \text{dom}(m_s)$, or $(i, \bar{\rho}, \kappa)$ with the same data, or a pair (i, ρ', j) , $(j, \bar{\rho}', i)$ with $i, j \in I(a)$ and $\rho' \notin \mathcal{R}^- \cup \mathcal{R}^+$. Only the first two forms are required to have an endpoint in $\text{dom}(m_s)$; the third may name an absent identifier, since its label is outside the ancestry vocabulary and Theorem 5.7 never inspects it.

S4 is the load-bearing clause. A derivation call may attach an ancestry-labelled edge only to an identifier it has just created. It may attach an edge to a pre-existing identifier only in the descendant direction, or with a label outside the ancestry vocabulary altogether. Everything in Section 5.2 follows from that clause.

Model object	ContextFS identifier at a93035d
Rec, the record set	<code>class Memory(BaseModel), schemas.py:1362</code>
Id, identifiers	<code>Memory.id, default str(uuid.uuid4())[:12], schemas.py:1371</code>
N , namespaces	<code>Memory.namespace_id, schemas.py:1384</code>
T , record types	<code>class MemoryType(str, Enum), schemas.py:23</code>
$\mathcal{R}$, relation labels	<code>class EdgeRelation(str, Enum), storage_protocol.py:32</code>
$\overline{(-)}$, the involution	<code>EdgeRelation.get_inverse, storage_protocol.py:80</code>
$\mathcal{R}^-$, ancestry labels	<code>EVOLVED_FROM, MERGED_FROM, SPLIT_FROM</code>
E_s , the edge set	<code>memory_edges</code> table, created by <code>migrations/versions/003_add_memory_edges.py:56-69</code> , written by <code>StorageRouter.add_edge, storage_router.py:1327</code>
S , stores	the <code>memories</code> table (<code>core.py:203-220</code>) together with the <code>memory_edges</code> table
α , the structure map	<code>class MemoryLineage, memory_lineage.py:61</code> , plus <code>ContextFS.save/recall</code>
save	<code>ContextFS.save, core.py:673</code> ; <code>StorageRouter.save, storage_router.py:89</code>
recall	<code>ContextFS.recall, core.py:1288</code> ; <code>StorageRouter.recall, storage_router.py:290</code>
evolve	<code>MemoryLineage.evolve, memory_lineage.py:93</code>
merge	<code>MemoryLineage.merge, memory_lineage.py:191</code>
split	<code>MemoryLineage.split, memory_lineage.py:360</code>
supersede	<code>MemoryLineage.supersede, memory_lineage.py:508</code>
link	<code>MemoryLineage.link, memory_lineage.py:551</code>
Θ , merge strategies	<code>class MergeStrategy(str, Enum), memory_lineage.py:43</code>
Ξ , change reasons	<code>class ChangeReason(str, Enum), types/versioned.py:44</code>
β_s , lineage observation	<code>StorageRouter.get_lineage, storage_router.py:1683</code>
$\llbracket i \rrbracket_s^-$, ancestry behaviour	approximated by <code>MemoryLineage.get_history, memory_lineage.py:599</code>
obs, timed observation	<code>Timeline.at, types/versioned.py:238</code>
abstract carrier signature	<code>class StorageBackend(Protocol), storage_protocol.py:148</code>
abstract edge signature	<code>class GraphBackend(Protocol), storage_protocol.py:347</code>

Table 1: Model objects and the ContextFS identifiers that realise them at commit a93035d.

4 The system

ContextFS is a memory layer for coding agents, written in Python. At commit a93035d its state lives in a SQLite database, with an optional vector store for similarity search and an optional graph database for edge queries. SQLite is the authoritative store on every write path (`storage_router.py`, `save`, line 89). Three surfaces expose the same operations: a Python facade (`core.py`, class `ContextFS`, line 34), a Model Context Protocol server (`mcp/fastmcp_server.py`) and a command line (`cli/memory.py`). Throughout this paper, ContextFS paths are relative to `src/contextfs/` in the repository at `/Users/mlong/Documents/Development/contextfs-ai/contextfs`, and line numbers are those of commit a93035d.

4.1 Model to code

Table 1 gives the correspondence. Every identifier in the right-hand column was read at a93035d before being listed.

4.2 The relation vocabulary

Listing 1 shows the head of the label enumeration. At a93035d it has twenty-two members, of which twenty appear in the inverse table of `EdgeRelation.get_inverse` and two, `DEPENDS_ON` and `IMPLEMENTS`, do not and therefore fall through the table’s default.

```
class EdgeRelation(str, Enum):
    # Evolution relationships
    EVOLVED_INT0 = "evolved_into"
    EVOLVED_FROM = "evolved_from"
    # Merge relationships
    MERGED_INT0 = "merged_into"
    MERGED_FROM = "merged_from"
    # Split relationships
    SPLIT_INT0 = "split_into"
    SPLIT_FROM = "split_from"
```

Listing 1: `src/contextfs/storage_protocol.py`, lines 32–49 with the class docstring and blank lines elided, commit a93035d. The first six of the twenty-two labels.

Lemma 4.1 (The inverse map is an involution). *Let $\overline{(-)}$ denote `EdgeRelation.get_inverse` at a93035d: the function that looks a label up in the table at lines 82 to 104 of `storage_protocol.py` and returns the label itself when it is absent. Then $\overline{\overline{\rho}} = \rho$ for every $\rho \in \mathcal{R}$, so $(\mathcal{R}, \overline{(-)})$ is an inverse structure in the sense of Theorem 3.2, with four fixed points.*

Proof. The table lists twenty entries. Eighteen of them form the nine unordered pairs

EVOLVED_INT0/EVOLVED_FROM	MERGED_INT0/MERGED_FROM	SPLIT_INT0/SPLIT_FROM
REFERENCES/REFERENCED_BY	SUPERSEDES/SUPERSEDED_BY	PARENT_OF/CHILD_OF
PART_OF/CONTAINS	CAUSED_BY/CAUSES	RESOLVES/RESOLVED_BY

each of which appears in the table in both orders, so $\overline{\overline{\rho}} = \rho$ on those eighteen. The remaining two table entries send `RELATED_TO` and `CONTRADICTS` to themselves. The two labels absent from the table, `DEPENDS_ON` and `IMPLEMENTS`, are returned unchanged by the default of the lookup. Every label is therefore either paired with a distinct label that is paired back with it, or fixed, and the map is an involution whose fixed-point set is $\{\text{RELATED_TO}, \text{CONTRADICTS}, \text{DEPENDS_ON}, \text{IMPLEMENTS}\}$. $\square$

Remark 4.2. The two implicit fixed points are not the same kind of object as the two explicit ones. `RELATED_TO` and `CONTRADICTS` are symmetric relations and are annotated as such in the source. `DEPENDS_ON` and `IMPLEMENTS` are directed relations that acquire a fixed point only because the lookup defaults to the identity on a missing key. The involution law holds, but for one of these pairs it holds by accident of the default rather than by design.

4.3 Derivation

Listing 2 is the core of `evolve`. Three features matter below. A new `Memory` is constructed rather than the original mutated. The ancestry is written into the new record’s `metadata` unconditionally. The two edges are written inside an exception handler that logs and continues.

```
evolved = Memory(
    content=new_content, type=original.type, tags=tags,
    namespace_id=original.namespace_id,
    metadata={**original.metadata,
```

```

        "evolved_from": memory_id,
        "evolution_timestamp": datetime.now(timezone.utc).isoformat(),
        "change_reason": reason.value})
self._storage.save(evolved)
try:
    self._storage.add_edge(from_id=evolved.id, to_id=memory_id,
                           relation=EdgeRelation.EVOLVED_FROM)
    self._storage.add_edge(from_id=memory_id, to_id=evolved.id,
                           relation=EdgeRelation.EVOLVED_INT0)
except Exception as e:
    logger.warning(f"Failed to create evolution edges: {e}")

```

Listing 2: `src/contextfs/memory_lineage.py`, lines 147–182 abridged, commit a93035d. Four constructor keyword arguments (`summary`, `source_repo`, `project`, `source_tool`) are elided.

`merge` and `split` follow the same shape: construct new records, write `merged_from` or `split_from` into `metadata`, then attempt the edge pairs inside a handler. `supersede` writes only edges, with the timestamp in the edge metadata.

Listing 3 shows the attribute selection performed by `merge` under the default union strategy. The namespace and type of the merged record are selected here and in the constructor call at line 256, and both selections are the subject of results in Section 5.

```

if strategy == MergeStrategy.UNION:
    all_tags = [tag for m in memories for tag in m.tags]
    combined_meta = {}
    for m in memories:
        combined_meta.update(m.metadata)
    types = [m.type for m in memories]
    final_type = max(set(types), key=types.count)
    ### line 256, in the merged record's constructor:
        namespace_id=memories[0].namespace_id,

```

Listing 3: `src/contextfs/memory_lineage.py`, two non-adjacent excerpts, commit a93035d: lines 298–307 (the union branch of `_apply_merge_strategy`, with its comments elided) and line 256 (the namespace argument of the merged record’s constructor). The two are separated below by an inserted marker line.

4.4 Observation

Two observation paths exist and they are not the same. `recall` resolves an identifier against SQLite by prefix (Listing 4). `Timeline.at` answers the timed question over an in-memory version list (Listing 5).

```

def _recall_from_sqlite(self, memory_id: str) -> Memory | None:
    conn = sqlite3.connect(self._db_path)
    cursor = conn.cursor()
    try:
        cursor.execute(
            "SELECT * FROM memories WHERE id LIKE ? AND deleted_at IS NULL",
            (f"{memory_id}%",),
        )
        row = cursor.fetchone()
        if row:
            return self._row_to_memory(row)
    return None

```

```

finally:
    conn.close()

```

Listing 4: `src/contextfs/storage_router.py`, lines 310–326 abridged, commit a93035d.

```

def at(self, timestamp: datetime) -> VersionEntry[S] | None:
    ts = timestamp
    if ts.tzinfo is None:
        ts = ts.replace(tzinfo=timezone.utc)
    result = None
    for entry in self.entries:
        entry_ts = entry.timestamp
        if entry_ts.tzinfo is None:
            entry_ts = entry_ts.replace(tzinfo=timezone.utc)
        if entry_ts <= ts:
            result = entry
        else:
            break
    return result

```

Listing 5: `src/contextfs/types/versioned.py`, lines 238–263 abridged, commit a93035d.

`Timeline.at` compares against a single timestamp field of `VersionEntry` (`types/versioned.py:117`). There is no second time field to compare against, which is the content of Theorem 5.27.

4.5 The agent-facing surface

Listing 6 shows the tail of the facade’s `evolve`. The `reason` parameter of `MemoryLineage.evolve` is not among the arguments forwarded, so the default `ChangeReason.OBSERVATION` is used. The Model Context Protocol tool (`mcp/fastmcp_server.py:623`) and the command line (`cli/memory.py:247`) call this same facade method and likewise do not expose a change reason.

```

if preserve_tags is None:
    preserve_tags = self.config.lineage_preserve_tags

return self._lineage.evolve(
    memory_id=memory_id,
    new_content=new_content,
    summary=summary,
    preserve_tags=preserve_tags,
    additional_tags=additional_tags,
)

```

Listing 6: `src/contextfs/core.py`, lines 1545–1554, commit a93035d.

5 Law checking

Throughout this section (S, α) is the lineage realisation given by ContextFS at a93035d under the correspondence of Table 1, with $\mathcal{R}^- = \{\text{EVOLVED_FROM}, \text{MERGED_FROM}, \text{SPLIT_FROM}\}$. Every claim about the code is a claim about the source at that commit.

5.1 L1, record monotonicity

Proposition 5.1 (Derivation calls are monotone). *Every derivation call satisfies L1: for $a \in D$ and any ω whose allocated identifiers lie outside $\text{supp}(s)$, writing $t = \text{nx}(s, a, \omega)$, we have $\text{dom}(m_s) \subseteq \text{dom}(m_t)$ and $m_t \upharpoonright_{\text{dom}(m_s)} = m_s$, while $|\text{dom}(m_t)| - |\text{dom}(m_s)|$ is 1 for evolve and merge, k for a k -part split, and 0 for supersede.*

Proof. Three of the four methods, `evolve`, `merge` and `split`, obtain their inputs through `self._storage.recall`, which returns a `Memory` object read out of SQLite. Each then constructs one or more new `Memory` instances with the default identifier factory (`schemas.py:1371`) and persists them with `self._storage.save` applied to a newly constructed instance (`memory_lineage.py:165, 269, 436`). The fourth, `supersede`, retrieves nothing, constructs nothing and issues no `save` at all (`memory_lineage.py`, lines 508 to 545). It writes two edges and returns, so it is monotone trivially. No method assigns to a field of a retrieved object, and none issues a write against the identifier of a retrieved object. Since the constructed identifiers are drawn fresh, the pre-existing part of m_s is untouched. The domain grows by exactly the number of constructed records, which is one for `evolve`, one for `merge`, `len(parts)` for `split` and zero for `supersede`. $\square$

Proposition 5.2 (Two operations outside D break monotonicity). *L1 is stated for derivation calls only, and the restriction is necessary: at [a93035d](#) the implementation has two write paths outside D that are not monotone. A `save` issued under an existing identifier replaces the stored record, and a `deletion` removes the record together with every edge incident to it in either direction.*

Evidence. For the first path, `StorageRouter._save_to_sqlite` issues `INSERT OR REPLACE INTO memories` (`storage_router.py:172`), and `ContextFS.save` accepts an explicit `id` argument (`core.py:686`). For the second, `StorageRouter.delete` (`storage_router.py:586`) resolves the identifier and calls `_delete_edges_for_memory`, which issues `DELETE FROM memory_edges WHERE from_id = ? OR to_id = ?` (`storage_router.py:626,639`), before deleting the record itself. Deletion is therefore hard rather than soft, on the record and on every incident edge. $\square$

Proposition 5.3 (Identifier freshness is assumed, not checked). *Every statement about derivation calls in this paper carries the hypothesis that the ambient parameter allocates identifiers fresh for the store. At [a93035d](#) nothing in the implementation establishes that hypothesis, and the write path converts a violation of it into a silent replacement rather than an error.*

Evidence. `Memory.id` has default factory `str(uuid.uuid4())[0:12]` (`schemas.py:1371`). The value is drawn at random and is not compared against the store: neither `MemoryLineage` nor `StorageRouter.save` (`storage_router.py:89`) queries for an existing row with that identifier before writing, and `_save_to_sqlite` issues `INSERT OR REPLACE INTO memories` (line 172). A drawn identifier that collides with a stored one therefore replaces that row, and the collision is not reported. $\square$

Remark 5.4 (The freshness hypothesis and where it is discharged). This is a hypothesis in the model, so it is worth being exact about its status. In Theorem 3.8 the allocated identifiers are a component of the ambient parameter, and Theorem 5.7 quantifies over parameters whose identifiers are fresh. That is a legitimate way to state a theorem, and the theorem is true as stated. What Theorem 5.3 adds is that the implementation does not restrict itself to such parameters: it samples them, so the hypothesis holds with high probability rather than by construction. The scale at which the probability stops being negligible is given in Theorem 6.8: roughly five million records for a collision between two identifiers. That is a far larger store than the one at which Theorem 6.3 bites. Read together, the two say that identifier discipline in `ContextFS` degrades in a specific order as a

store grows: abbreviated lookups become ambiguous first, by about two orders of magnitude, and outright allocation collisions much later. A realisation that checked freshness on write, or that drew identifiers from a counter, would discharge the hypothesis and lose nothing else.

Remark 5.5. The deletion path has a further consequence. Because `_delete_edges_for_memory` removes edges in which the deleted memory is either endpoint, deleting an ancestor erases that ancestry relation from every descendant that pointed at it. The resulting store remains ancestry-closed: the incident edges disappear with the record. What fails is cross-state preservation of previously recorded ancestry, including for descendants the caller did not name. Deletion is the one operation in the API that can erase this part of the past.

5.2 Stable ancestry

L1 says records survive. It does not say that what can be observed about a record survives, and in general it does not: `evolve` adds an `EVOLVED_INT0` edge out of the original, so the original's outgoing edge set changes. The content of the following theorem is that the change is confined to the descendant direction.

Lemma 5.6 (ContextFS instantiates the schema). *For every ambient parameter whose allocated identifiers are fresh for the store, the lineage realisation given by ContextFS at `a93035d` follows the derivation schema of Theorem 3.23. This is an instantiation claim in the sense of Section 1.3, established by reading the four methods of `src/contextfs/memory_lineage.py`. The freshness proviso is not vacuous and is discussed in Theorem 5.3.*

Evidence. S1: `evolve` and `split` call `self._storage.recall` on their single named identifier and raise `ValueError` when it returns `None` (lines 132 to 134, 397 to 399). `merge` loops over its named identifiers, appends those that resolve and logs a warning for those that do not, then raises when fewer than two resolve (lines 225 to 232). `supersede` constructs no record and performs no retrieval at all: it calls `add_edge` directly on its two named identifiers (lines 525 to 540), which is the case S1's last sentence allows, and which is sound here only because both labels it writes lie outside $\mathcal{R}^- \cup \mathcal{R}^+$. No method reads a record it did not name.

S2 and S3: each of `evolve`, `merge` and `split` constructs `Memory` instances with the default identifier factory (`schemas.py:1371`) and persists them with `self._storage.save` (lines 165, 269, 436); `supersede` constructs none. S2 also requires the constructed identifiers to be the ones the ambient parameter allocates, which is the freshness hypothesis; the implementation samples rather than checks it, and the exact status of that hypothesis is Theorem 5.3 and Theorem 5.4. The lemma is therefore stated for ambient parameters whose allocated identifiers are fresh for the store, as are all the results that use it. The namespace of each constructed record is copied from a retrieved record: `namespace_id=original.namespace_id` in `evolve` (line 152) and `split` (line 422), `namespace_id=memories[0].namespace_id` in `merge` (line 256).

S4: the added triples are, for `evolve`, $(\kappa, \text{EVOLVED_FROM}, i)$ and $(i, \text{EVOLVED_INT0}, \kappa)$ with κ the constructed identifier (lines 170 to 180); for `merge`, $(\kappa, \text{MERGED_FROM}, i_r)$ and $(i_r, \text{MERGED_INT0}, \kappa)$ for each retrieved i_r (lines 275 to 285); for `split`, $(\kappa_r, \text{SPLIT_FROM}, i)$ and $(i, \text{SPLIT_INT0}, \kappa_r)$ for each constructed κ_r (lines 442 to 452); and for `supersede`, $(j, \text{SUPERSEDES}, i)$ and $(i, \text{SUPERSEDED_BY}, j)$ between two named identifiers (lines 526 to 540), with `SUPERSEDES` and `SUPERSEDED_BY` outside $\mathcal{R}^- \cup \mathcal{R}^+$. No method removes an edge. The labels used are inverse to one another by Theorem 4.1.

The point on which Theorem 5.7 turns is best made here rather than left to the proof. In each of the first three methods the two added edges share their pair of endpoints and differ only in direction and label. The label pointing from the pre-existing identifier to the constructed one is always the $\mathcal{R}^+$ member of the pair, never the $\mathcal{R}^-$ member: `EVOLVED_INT0` rather than `EVOLVED_FROM` out of

memory_id, MERGED_INT0 rather than MERGED_FROM out of each original.id, SPLIT_INT0 rather than SPLIT_FROM out of memory_id. Since $\mathcal{R}^- \cap \mathcal{R}^+ = \emptyset$ by Theorem 3.3, no edge with a label in $\mathcal{R}^-$ acquires a pre-existing source, which is clause S4 and what the theorem uses. $\square$

Theorem 5.7 (Ancestry immutability). *Let (S, α) be any lineage realisation following the derivation schema. Let s be an ancestry-closed store, let $a \in D$ and let ω be an ambient parameter whose identifiers are fresh for s . Put $t = \text{nx}(s, a, \omega)$ and $D_0 = \text{dom}(m_s)$. Then t is ancestry-closed, β_s^- restricts to a coalgebra structure on D_0 , and the inclusion $\iota: D_0 \hookrightarrow \text{Id}$ is a homomorphism of H^- -coalgebras*

$$\iota: (D_0, \beta_s^- \upharpoonright_{D_0}) \longrightarrow (\text{Id}, \beta_t^-).$$

Proof. Write $\Delta = E_t \setminus E_s$, which is well defined and satisfies $E_s \subseteq E_t$ by S4.

Claim: every triple in Δ whose label lies in $\mathcal{R}^-$ has its source outside D_0 . By S4 a triple in Δ is of one of three forms. In the first its label lies in $\mathcal{R}^-$ and its source is an identifier allocated by ω , hence fresh for s and so not in D_0 . In the second its label lies in $\mathcal{R}^+$, which is disjoint from $\mathcal{R}^-$ by Theorem 3.3. In the third its label lies outside $\mathcal{R}^- \cup \mathcal{R}^+$. Only the first form contributes a triple with label in $\mathcal{R}^-$, and it has source outside D_0 .

Ancestry closure of t . Let $i \in \text{dom}(m_t)$ and $(i, \rho, j) \in E_t$ with $\rho \in \mathcal{R}^-$. If the triple lies in E_s then $i \in D_0$, since an identifier in $\text{dom}(m_t) \setminus D_0$ is fresh for s by S2 and so occurs in no triple of E_s ; ancestry closure of s then gives $j \in D_0 \subseteq \text{dom}(m_t)$ by S2. If instead the triple lies in Δ then by the claim it has the first form of S4, whose target lies in $I(a) \cap \text{dom}(m_s) \subseteq D_0 \subseteq \text{dom}(m_t)$.

β_s^- restricts. For $i \in D_0$, ancestry closure of s gives that every $\mathcal{R}^-$ -labelled edge out of i lands in D_0 , so $\beta_s^-(i) \in \text{Rec}_\perp \times \mathcal{P}_{\text{fin}}(\mathcal{R}^- \times D_0)$ and the restriction is a map $D_0 \rightarrow H^-D_0$.

The homomorphism condition. Let $i \in D_0$. We must show $\beta_t^-(\iota(i)) = H^-\iota(\beta_s^-(i))$. The first components agree because m_t extends m_s by S2. Since $H^-\iota$ acts as the identity on labels and as ι on targets, the second components agree exactly when

$$\{(\rho, j) : (i, \rho, j) \in E_t, \rho \in \mathcal{R}^-\} = \{(\rho, j) : (i, \rho, j) \in E_s, \rho \in \mathcal{R}^-\}.$$

The inclusion from right to left is $E_s \subseteq E_t$. For the other, a triple $(i, \rho, j) \in E_t \setminus E_s$ with $\rho \in \mathcal{R}^-$ lies in Δ , so by the claim $i \notin D_0$, contradicting $i \in D_0$. Hence ι is a homomorphism. $\square$

Corollary 5.8. *The conclusion of Theorem 5.7 holds of ContextFS at a93035d, conditionally on Theorem 5.6.*

The hypothesis of ancestry closure needs justifying rather than assuming, because Theorem 5.30 below shows that L7 fails: not every store of S is ancestry-closed. What rescues the theorem is that ancestry closure is an invariant of the fragment the theorem is about, so the relevant stores are not merely hypothetical.

Proposition 5.9 (Ancestry closure is reachable and is an invariant). *For any realisation following the derivation schema, the empty store is ancestry-closed. Every derivation call takes an ancestry-closed store to an ancestry-closed store, and so does every call that adds a record with a fresh identifier and no edges. Consequently every store reachable from the empty store by such calls is ancestry-closed, and Theorem 5.7 applies at every step of every such run.*

Proof. The empty store has no edges, so it is vacuously ancestry-closed. Preservation under derivation calls is the first conclusion of Theorem 5.7. A call that adds a record at a fresh identifier and no edges leaves E unchanged and enlarges $\text{dom}(m)$, and ancestry closure is monotone in $\text{dom}(m)$ for fixed E . Induction on the length of the run. $\square$

Proposition 5.10 (The unchecked escape route). *At a93035d, `MemoryLineage.link` is a code path outside the fragment of Theorem 5.9 that can produce a store that is not ancestry-closed.*

Evidence. `MemoryLineage.link` (`memory_lineage.py:551`) accepts an arbitrary `relation` argument, including an ancestry label, and calls `add_edge` without retrieving either endpoint. `add_edge` takes `validate: bool = False` (`storage_router.py:1327`) and the call site omits the argument. The facade `ContextFS.link` (`core.py:1798`) is not such a path: it recalls both endpoints and returns `False` when either is absent (lines 1827 to 1831), then passes the resolved full identifiers. $\square$

Remark 5.11. `StorageRouter.delete` does not supply a second route. It resolves the identifier twice, once before deleting incident edges and once before deleting the row. The memory set is unchanged between those queries, so the fixed resolution parameter π of Theorem 3.8 selects the same row both times. An ambiguous prefix still leaves the identity of the deleted row dependent on an order the interface does not expose, as in Theorem 6.3, but it does not leave a dangling ancestry edge in this model.

Corollary 5.12 (Ancestry behaviour is invariant). *Under the hypotheses of Theorem 5.7, $\llbracket i \rrbracket_t^- = \llbracket i \rrbracket_s^-$ for every $i \in \text{dom}(m_s)$: the ancestry unfolding of a memory is unchanged by any later derivation call. Consequently, for any $i, i' \in \text{dom}(m_s)$, i and i' are H^- -bisimilar in s if and only if they are H^- -bisimilar in t .*

Proof. Behaviour maps into a final coalgebra compose with homomorphisms: since ι is a homomorphism and $\llbracket - \rrbracket_t^-$ is the unique homomorphism out of (Id, β_t^-) , the composite $\llbracket - \rrbracket_t^- \circ \iota$ is a homomorphism out of $(D_0, \beta_s^- \upharpoonright_{D_0})$, as is the restriction of $\llbracket - \rrbracket_s^-$. By finality these coincide, which is the displayed equality. The bisimilarity statement follows because H^- preserves weak pullbacks (Theorem 3.17), so bisimilarity is equality of behaviour. $\square$

Proposition 5.13 (Descendant structure is not stable). *Theorem 5.7 fails for H in place of H^- . Concretely, let s contain a single record at i with no edges, let $a = \text{evolve}(i, c, \xi)$ and let ω allocate κ . Then $\beta_s(i)$ has empty edge component while $\beta_t(i)$ has edge component $\{(\text{EVOLVED_INTO}, \kappa)\}$, so ι is not an H -homomorphism and $\llbracket i \rrbracket_t \neq \llbracket i \rrbracket_s$.*

Proof. Immediate from the enumeration of added edges in the proof of Theorem 5.7: the call adds $(i, \text{EVOLVED_INTO}, \kappa)$ with source $i \in D_0$. The two behaviours differ because a state with no outgoing transitions is not bisimilar to a state with one. $\square$

The asymmetry is the useful content. A consumer of `ContextFS` may cache the ancestry of a memory and rely on it. It may not cache the descendants. In the terms of the series, this is the structural property the memory layer contributes to any composite that contains it, and Section 6 states what a skill or a harness is entitled to conclude from it.

5.3 L2, edge and record agreement

Proposition 5.14 (Ancestry is recorded twice and the two records can disagree). *L2 fails. The implementation at a93035d maintains two independent representations of ancestry, and the write of one is guarded while the write of the other is not.*

Proof. In `evolve`, the key `"evolved_from"` is written into the constructed record's metadata dictionary as part of the `Memory(...)` constructor call (`memory_lineage.py:158`), and the record is then persisted by an unguarded `self._storage.save` (line 165). The two `EVOLVED_FROM`/`EVOLVED_INTO` edges are written afterwards inside `try: ... except Exception as e:`

`logger.warning(...)` (lines 168 to 182). Any exception raised by `add_edge`, including one raised by the first of the two calls before the second is attempted, is caught, logged at warning level, and not re-raised. Control then falls through to `return evolved`. The same structure appears in `merge` (metadata at line 262, guarded edges at lines 273 to 287) and in `split` (metadata at lines 426 to 432, guarded edges at lines 440 to 454). Hence a store is reachable in which $m_s(\kappa)$ carries `metadata["evolved_from"] = i` while $(\kappa, \text{EVOLVED_FROM}, i) \notin E_s$, which is a violation of L2. $\square$

Remark 5.15 (The two representations are read by different code paths). The divergence is not merely latent, because both representations are read. `StorageRouter.get_lineage` (`storage_router.py:1683`) first delegates to the graph backend if one is configured, then queries the edge table. If the edge query yields neither ancestors nor descendants, it falls back to reading `evolved_from`, `merged_from` and `split_from` out of the record's metadata (lines 1712 to 1739). `MemoryLineage.get_history` (`memory_lineage.py:599`) calls that method and then applies a second, and differently shaped, metadata fallback of its own (lines 639 to 663). The two fallbacks do not agree on the shape of an ancestor entry: the router emits dictionaries keyed "id", "relation", "depth" (line 1274), while `get_history` emits dictionaries keyed "memory_id" and "relation" with no depth (lines 641 to 646). A consumer of the lineage observation therefore cannot rely on a single field name for the ancestor identifier, which is a well-typedness failure of the observation and not only a redundancy failure.

5.4 L3, inverse closure

Proposition 5.16 (Inverse closure holds on the unexceptional path). *Every derivation call at `a93035d` writes edges in inverse pairs, so along any execution in which no `add_edge` call raises, every reachable store is inverse-closed. The property is not maintained unconditionally: because the two writes of a pair are separate calls inside one handler, an exception raised by the second leaves the first in place.*

Proof. The enumeration in the proof of Theorem 5.7 lists, for each derivation call, the added triples in $(-)$ -inverse pairs, and Theorem 4.1 shows the labels used are genuinely inverse to one another. `link` writes the inverse only when its `bidirectional` argument is true, and then obtains the inverse label through `EdgeRelation.get_inverse` (`memory_lineage.py`, lines 580 to 587). So `link` calls with $b = 0$ do not preserve inverse closure, which is why L3 is stated for derivation calls. For the failure clause, the two `add_edge` calls of a pair are consecutive statements inside a single `try` block (for instance `memory_lineage.py`, lines 170 to 180), so an exception in the second is caught by the same handler that would have caught one in the first, and the effect of the first is not undone. $\square$

5.5 L4, namespace locality

The two clauses of L4 come apart, and which one holds is the useful information.

Proposition 5.17 (Confinement holds). *$L_4(i)$ holds for derivation calls under resolution by equality. Let $\nu \in N$, let s be a store, let $a \in D \cap A_\nu(s)$ and let ω be an ambient parameter allocating identifiers fresh for s . Then $\alpha(\pi_\nu s)(a, \omega) = (\text{id} \times \pi_\nu)(\alpha(s)(a, \omega))$.*

Proof. Every identifier named in a lies in $\text{dom}(m_{\pi_\nu s})$ by the definition of $A_\nu(s)$, and the record stored at each is the same in s and $\pi_\nu s$, since π_ν restricts the domain and does not alter values. Under resolution by equality the three retrieving methods therefore obtain the same records in both stores, so they raise in neither. The records they construct are equal, since each constructed record

is a function of the retrieved records, the non-identifier arguments and ω . `supersede` retrieves nothing, and its output is a function of its two named identifiers alone. Outputs therefore agree.

For the state component we must check that the added data survives restriction to ν . Each constructed record takes its namespace from a named argument: `evolve` and `split` set `namespace_id=original.namespace_id` (`memory_lineage.py:152, 422`) and `merge` sets `namespace_id=memories[0].namespace_id` (line 256). Since every named argument lies in the fibre $\text{ns}^{-1}(\nu)$, every constructed record does too, so it is retained by π_ν . Each added edge is incident to one named identifier and one constructed identifier, both now in the fibre, so it is retained by the edge clause of Theorem 3.22. `supersede` adds no record and two edges between named identifiers, both in the fibre. Hence $\text{nx}(\pi_\nu s, a, \omega) = \pi_\nu(\text{nx}(s, a, \omega))$. $\square$

Proposition 5.18 (Construction locality fails). *L4(ii) fails. There is a store s , a call a and an ambient parameter ω such that the record constructed by $\alpha(s)(a, \omega)$ lies in the fibre $\text{ns}^{-1}(\nu)$ while its payload depends on a record of s lying in a different fibre.*

Proof. Let s contain records at i_1, i_2 with $\text{ns} = \nu$ and at i_3 with $\text{ns} = \nu' \neq \nu$, where the record at i_3 carries a metadata key k absent from the other two and content distinct from theirs. Let $a = \text{merge}(\langle i_1, i_3, i_2 \rangle, \perp, \text{UNION})$. The retrieval loop (`memory_lineage.py`, lines 225 to 230) resolves all three identifiers, and the guard at line 232 passes. The constructor at line 256 sets `namespace_id=memories[0].namespace_id = \nu`, so the constructed record lies in the fibre $\text{ns}^{-1}(\nu)$. Its content is produced by `_generate_merged_content` (lines 349 to 354), which concatenates a truncation of each argument's content in argument order, so it contains the content of the record at i_3 . Its metadata is produced by the union branch of `_apply_merge_strategy`, whose loop `combined_meta.update(m.metadata)` (lines 303 to 304) runs over every retrieved record, so it contains k . The payload of a record in the fibre $\text{ns}^{-1}(\nu)$ therefore depends on a record outside that fibre, which is a violation of L4(ii). No comparison of the arguments' namespaces occurs anywhere in `merge`. $\square$

Remark 5.19 (What the two clauses say together). The pair of results is more informative than either alone. Confinement (Theorem 5.17) says the boundary is not leaky. An operation that stays inside a namespace has effects that stay inside it, and no restricted view of the store can be surprised by an operation issued against the full store from within its own fibre. Construction locality (Theorem 5.18) says the boundary is nevertheless crossable, and by construction rather than by leakage. The field `namespace_id` is what `delete_by_namespace` (`storage_router.py:736`) and the namespace-scoped search paths select on, and `merge` files a record under the namespace of its first argument while copying content and metadata from all of them. A caller who merges across the boundary moves payload across it. This is an implementation-level statement about the source at `a93035d`, not a report of an exploited path, and the repair is a namespace comparison in the retrieval loop.

5.6 L5, resolution independence

Proposition 5.20 (The merged type reads the resolution parameter). *L5 fails. The `type` field of the record constructed by `merge` is computed as `max(set(types), key=types.count)` (`memory_lineage.py:307`, and identically at lines 323 and 345 for the intersection and weighted branches). When two distinct types attain the same maximal multiplicity in `types`, this expression returns whichever of the maximisers occurs first in the enumeration order of the Python set `set(types)`. In the model that order is a component of the resolution parameter π and not of the allocation part of ω .*

Proof. `max` with a `key` returns the first element of its iterable attaining the maximal key value, so when the multiset `types` has two or more modes the result is determined by the enumeration order of `set(types)`. The smallest witness has two arguments of distinct types: with `types == [DECISION, ERROR]` both members have count one, both are maximisers, and the returned value is the first member enumerated. Two narrowings are worth stating, since neither rescues the law. The expression occurs in the union, intersection and weighted branches only. The latest and oldest branches take the type of a designated argument and are unaffected. And a caller who supplies the optional `memory_type` argument overrides the computed value at `memory_lineage.py`, lines 247 to 248, which is evaluated after `_apply_merge_strategy` returns. So the failure is exhibited by any call that omits `memory_type` and uses one of the three affected strategies, the default among them. Take $\omega = (\vec{k}, t, \pi)$ and $\omega' = (\vec{k}, t, \pi')$ where π orders this two-element set one way and π' the other. Then $\alpha(s)(a, \omega)$ and $\alpha(s)(a, \omega')$ construct records differing in their `type` field, so they differ, contradicting L5. $\square$

Remark 5.21 (The order is not merely unspecified, it varies). The proof above needs only that the enumeration order is not fixed by the arguments. What fixes it in practice matters too, because the two readings have different consequences for a caller. `MemoryType` is declared `class MemoryType(str, Enum)` (`schemas.py:23`), and the enumeration order of a Python set of such members is determined by their hash values. The hash of a string-valued object is salted per process by default, and the salt is controlled by the `PYTHONHASHSEED` environment variable [13]. So π is not merely unspecified in the documentation. It is a parameter of the operating-system environment, and two processes running the same code against the same store may construct records with different types from the same call.

Remark 5.22. This is a small defect with a large formal consequence. L5 asks that a realisation read only the ambient data its own interface declares: fresh identifiers and the clock, both of which appear as record fields. A field decided by a hash-order tie-break is read from a parameter the interface never mentions. The repair is one line: replace the expression by a mode computation with a declared tie-break, for instance the type of the first argument, which would make `merge` agree with its own treatment of `namespace_id`. That change repairs this merge-local counterexample, not L5 for the whole action alphabet. Prefix recall also depends on an unordered row choice (Theorem 6.3); full repair requires deterministic identifier resolution, either exact matching or a documented total order.

5.7 Merge up to bisimilarity

L5 concerns one field of the merged record. The wider question is what algebraic laws `merge` obeys. The obvious answer, that it obeys none since every call allocates a fresh identifier, is an artefact of comparing states on the nose. The right comparison for a coalgebra is behavioural, and under it one of the three candidate laws holds.

Definition 5.23 (Allocation-blind observation). Let $q: \text{Rec} \rightarrow \text{Rec}/\approx$ be the quotient forgetting the `id`, `created_at` and `updated_at` fields and the metadata keys `evolution_timestamp`, `merge_timestamp` and `split_timestamp`. Let $H^\approx X = (\text{Rec}/\approx)_\perp \times \mathcal{P}_{\text{fin}}(\mathcal{R} \times X)$ and let $\beta_s^\approx = (q_\perp \times \text{id}) \circ \beta_s$, an $H^\approx$ -coalgebra on `Id`. Write $\llbracket i \rrbracket_s^\approx$ for the corresponding behaviour.

Proposition 5.24 (Merge is idempotent up to bisimilarity). *Let $a = \text{merge}(\vec{t}, c, \theta)$ be a call whose arguments all resolve in s , let $t_1 = \text{nx}(s, a, \omega_1)$ allocating κ_1 , and let $t_2 = \text{nx}(t_1, a, \omega_2)$ allocating $\kappa_2 \neq \kappa_1$, where ω_1 and ω_2 agree on clock and resolution parts. Then $\llbracket i \rrbracket_{t_1}^\approx = \llbracket i \rrbracket_{t_2}^\approx$ for every $i \in \text{dom}(m_{t_1})$.*

Proof. Write $\vec{i} = \langle i_1, \dots, i_n \rangle$ and let $\mathcal{B} = \{(i, i) : i \in \text{Id} \setminus \{\kappa_2\}\} \cup \{(\kappa_1, \kappa_2)\}$. We check that $\mathcal{B}$ is an $H^\approx$ -bisimulation between $(\text{Id}, \beta_{t_1}^\approx)$ and $(\text{Id}, \beta_{t_2}^\approx)$.

First note what the second call changes. By Theorem 5.1 it adds one record, at κ_2 , and by the enumeration in the proof of Theorem 5.7 it adds the edges $(\kappa_2, \text{MERGED_FROM}, i_r)$ and $(i_r, \text{MERGED_INTO}, \kappa_2)$ for $1 \leq r \leq n$, and nothing else. The second call resolves the same identifiers to the same records, and `merge`'s constructed record is a function of the retrieved records, the non-identifier arguments and ω . So the records at κ_1 and κ_2 agree in every field except `id`, `created_at`, `updated_at` and `merge_timestamp`, all of which q forgets. Hence $q(m_{t_1}(\kappa_1)) = q(m_{t_2}(\kappa_2))$.

Now take the pairs in turn. For (i, i) with $i \notin \{i_1, \dots, i_n\}$ we have $\beta_{t_1}^\approx(i) = \beta_{t_2}^\approx(i)$, since no record or edge at such an i was touched, and the successors are related by the identity part of $\mathcal{B}$. For (i_r, i_r) the record is unchanged, and the outgoing edge sets are $X \cup \{(\text{MERGED_INTO}, \kappa_1)\}$ on the left and $X \cup \{(\text{MERGED_INTO}, \kappa_1), (\text{MERGED_INTO}, \kappa_2)\}$ on the right, for a common X . Every left successor is $\mathcal{B}$ -related to a right successor with the same label, by the identity. Every right successor is $\mathcal{B}$ -related to a left one, using $(\kappa_1, \kappa_2) \in \mathcal{B}$ for the new pair. For (κ_1, κ_2) the quotiented records agree by the previous paragraph, and the outgoing edge sets are $\{(\text{MERGED_FROM}, i_r)\}_r$ on both sides, related by the identity. This exhausts the pairs, so $\mathcal{B}$ is a bisimulation. Since $H^\approx$ preserves weak pullbacks by the argument of Theorem 3.17, bisimilarity is equality of behaviour, and every $i \in \text{dom}(m_{t_1})$ is $\mathcal{B}$ -related to itself. $\square$

Proposition 5.25 (Merge is not commutative, even up to bisimilarity). *There are a store s , identifiers i_1, i_2 and an ambient parameter ω such that the record constructed by `merge`($\langle i_1, i_2 \rangle, \perp, \text{UNION}$) and the record constructed by `merge`($\langle i_2, i_1 \rangle, \perp, \text{UNION}$) have different images under q .*

Proof. Let the records at i_1 and i_2 lie in distinct namespaces $\nu_1 \neq \nu_2$. The constructor at `memory_lineage.py:256` takes `memories[0].namespace_id`, so the two calls produce records with namespaces ν_1 and ν_2 respectively, and q does not forget `namespace_id`. There is a second, independent witness. Even within one namespace, `_generate_merged_content` (lines 349 to 354) emits the argument contents in argument order under the labels `[Source 1]`, `[Source 2]` and so on. The two calls therefore produce different `content` whenever the two arguments have different content, and q does not forget `content`. $\square$

Remark 5.26. Associativity is not among the candidates, because `merge` is a variadic operation on a sequence of identifiers rather than a binary operation on records, and there is no identifier-level composite to associate. What the two propositions leave is a merge that is idempotent up to allocation-blind bisimilarity and sensitive to argument order in two independent ways. One of those ways, the namespace, is the same field responsible for the failure of L4(ii).

5.8 L6, bitemporality

Theorem 5.27 (The recorded lineage does not determine a bitemporal state). *Let Ψ record a bitemporal history h (Theorem 2.2) through the `ContextFS` lineage API at `a93035d`. Fix ambient parameters $\omega_1, \dots, \omega_n$ whose clock components are $t_1, \dots, t_n$, whose allocated identifiers $i_1, \dots, i_n$ are fresh at each step, and whose resolution components agree. The recording issues `save`(v_1) under ω_1 and `evolve`(i_{k-1}, v_k, ξ_k) under ω_k for $2 \leq k \leq n$. Then:*

1. *There are histories $h_a \neq h_b$, agreeing in every value v_k , every transaction time t_k and every change reason ξ_k , and differing only in valid times, with $\Psi(h_a) = \Psi(h_b)$ and $\sigma_{h_a} \neq \sigma_{h_b}$.*
2. *Consequently there is no function g on stores with $g(\Psi(h)) = \sigma_h$ for all h , and no bitemporal observation in the sense of Theorem 3.20 is sound for Ψ . The quantifier over g ranges over*

all functions of the entire store, so no reading of any record field, metadata key, edge, edge attribute, or derived timeline recovers valid time.

Proof. (1) Take $\text{Val} = \{A, B\}$, $t_1 = 1$ February, $t_2 = 1$ March, and

$$\begin{aligned} h_a &= ((A, 1 \text{ Jan}, 1 \text{ Feb}), (B, 10 \text{ Feb}, 1 \text{ Mar})), \\ h_b &= ((A, 1 \text{ Jan}, 1 \text{ Feb}), (B, 20 \text{ Feb}, 1 \text{ Mar})), \end{aligned}$$

with the same change reason on both second records. Evaluate Theorem 2.2 at $(\tau_v, \tau_t) = (15 \text{ Feb}, 15 \text{ Mar})$. For h_a both records satisfy $t_j \leq 15 \text{ Mar}$ and both satisfy $p_j \leq 15 \text{ Feb}$, so $J_{h_a} = \{1, 2\}$; the maximal valid-time start in J_{h_a} is $P = 10 \text{ Feb}$, attained only at $j = 2$, so $k = 2$ and $\sigma_{h_a}(\tau_v, \tau_t) = B$. For h_b we have $p_2 = 20 \text{ Feb} > 15 \text{ Feb}$, so $J_{h_b} = \{1\}$, $P = 1 \text{ Jan}$, $k = 1$ and $\sigma_{h_b}(\tau_v, \tau_t) = A$. Hence $\sigma_{h_a} \neq \sigma_{h_b}$. The two histories describe genuinely different situations: in h_a the value became B on 10 February and we found out on 1 March, while in h_b it became B on 20 February and we found out on 1 March. On 15 February the first was already B while the second was still A .

Now compare $\Psi(h_a)$ and $\Psi(h_b)$. The valid times p_k occur in no argument position of any operation of Theorem 3.7 as realised at a93035d. The signature of `MemoryLineage.evolve` (`memory_lineage.py`, lines 93 to 101) is `(memory_id, new_content, summary, preserve_tags, additional_tags, reason)`, and the record type `Memory` (`schemas.py`, lines 1362 to 1398) has exactly two time-valued fields, `created_at` and `updated_at`, both defaulting to `datetime.now()`. The metadata key written by `evolve` is "evolution_timestamp", assigned `datetime.now(timezone.utc).isoformat()` at line 159, again a reading of the clock at the moment of the call. Therefore the two recordings issue the same calls with the same arguments at the same transaction times under the same allocation parameter, and $\Psi(h_a) = \Psi(h_b)$.

(2) If such a g existed then $\sigma_{h_a} = g(\Psi(h_a)) = g(\Psi(h_b)) = \sigma_{h_b}$, contradicting (1). Since g was an arbitrary function on stores, no quantity computed from the store, however derived, determines σ_h . The soundness claim is the same argument applied to a bitemporal observation `obs`: at the common store $\Psi(h_a) = \Psi(h_b)$ and the common arguments $(\ell, 15 \text{ Feb}, 15 \text{ Mar})$ it would have to return B and A , which no function does. $\square$

The result is stronger than counting timestamp fields. The store contains `created_at`, `updated_at`, several timestamp metadata keys, timestamps on `VersionEntry`, and a temporal ordering implicit in the ancestry graph. The theorem quantifies over every function of the whole store, so none of those representations can recover valid time for the recording map Ψ .

Corollary 5.28 (`Timeline.at` is a transaction-time query). *`Timeline.at` (`types/versioned.py:238`) computes the last version entry whose single timestamp field is at or before the argument. When the timeline is built by `VersionedMem.from_memory` (`types/versioned.py:359`), that field is set from `memory.created_at` (line 387). When it is built by `VersionedMem.from_lineage` (line 402), it is read out of lineage history entries whose timestamps are likewise clock readings at call time. In the vocabulary of Section 2.3, `Timeline.at` answers “what did the system have on file at t ” and not “what was the case at t ”.*

Remark 5.29 (The weaker distinction is also not exposed). Theorem 5.27 is deliberately independent of the `ChangeReason` parameter, since the two histories it constructs carry the same reason. The parameter is in any case not reachable from outside the lineage module at a93035d. `ChangeReason` has four members, `OBSERVATION`, `INFERENCE`, `CORRECTION` and `DECAY` (`types/versioned.py`, lines 63 to 71), and `MemoryLineage.evolve` accepts it with default `ChangeReason.OBSERVATION` (line

100). None of the three agent-facing surfaces forwards it. `ContextFS.evolve` (`core.py:1515`) calls the lineage method with five keyword arguments, none of them `reason` (Listing 6). Both `contextfs_evolve` (`mcp/fastmcp_server.py:623`) and the command line `evolve` (`cli/memory.py:247`) call `ContextFS.evolve`. So on every agent-facing path the recorded reason is `OBSERVATION`, and a correction is not distinguished from an observation even by the four-valued label, let alone by a time.

5.9 L7, lineage referential integrity

Proposition 5.30 (Edge endpoints are not checked by default). *L7 fails as a property of the whole state space. Stores that are not ancestry-closed are reachable through the path of Theorem 5.10, and no constraint in the storage layer prevents them.*

Evidence. `StorageRouter.add_edge` (`storage_router.py:1327`) takes a parameter `validate: bool = False` and performs the endpoint existence query only when it is true (lines 1354 to 1370). Its documentation states that the default is false “since most callers (`core.link`, `memory_lineage`) already validate via `recall()`”. Every call site in `memory_lineage.py` omits the argument. The `memory_edges` table is created without a foreign key constraint on either endpoint. Reachability of a violating store is Theorem 5.10. $\square$

Remark 5.31 (What the failure of L7 does and does not cost). The documented justification is accurate for the three retrieving derivation methods, which do resolve every named identifier through `recall` before writing edges, and it is accurate for the facade `link`. It is not a property of the store because the direct `MemoryLineage.link` path in Theorem 5.10 omits the check.

The consequence for the rest of the paper is bounded, and Theorem 5.9 is what bounds it. Ancestry closure is not an accident of a well-chosen initial state: it holds at the empty store and is preserved by every derivation call and every fresh save, so it holds throughout any run built from those calls and the facade `link`. The results of Section 5.2 therefore describe the reachable behaviour of the system on its own principal interface, and the hypothesis they carry is discharged for that fragment rather than assumed. What L7’s failure costs is that the fragment is not the whole state space: a caller that reaches past the facade into `MemoryLineage.link` can leave it. Deletion also lies outside the monotone derivation fragment, but it removes incident edges and does not by itself violate ancestry closure. Outside that fragment Theorem 5.7 has nothing to say.

5.10 Summary

Table 2 collects the results. The passing clauses are enforced by the shape of the code, and what fails would require a check the code does not perform. Monotonicity holds because the derivation methods construct new objects and never assign to retrieved ones; inverse closure holds because the two writes are adjacent statements; confinement holds because every constructed record takes its namespace from a named argument. Construction locality, resolution independence, bitemporality and referential integrity would each require an explicit comparison, an explicit tie-break, an extra field, or a constraint, and none is present. Counted by clause, three hold and five fail.

6 Composition with adjacent layers

6.1 Layer contribution

A composite that contains a memory coalgebra gains one structural property that is not available from a stateless component: an observation that survives further activity. The following corollary

Law	Status at a93035d	Witness
L1 monotonicity	holds on D fails outside D	S2 of the schema; Theorem 5.1 save with an explicit id; delete (Theorem 5.2)
L2 edge/record agreement	fails	Theorem 5.14
L3 inverse closure	holds on the unexceptional path	Theorem 5.16
L4(i) confinement	holds on D	S3 and S4 of the schema; Theorem 5.17
L4(ii) construction locality	fails	Theorem 5.18
L5 resolution independence	fails	Theorem 5.20
L6 bitemporality	fails	Theorem 5.27
L7 referential integrity	fails	Theorem 5.30

Table 2: The eight law clauses of Theorem 3.21 against ContextFS at commit a93035d. Positive entries are theorems of Section 3 applied through Theorem 5.6; negative entries are counterexamples with a named code witness.

states the form a consumer can use.

Corollary 6.1 (Cacheable observation). *Let s_0 be an ancestry-closed store and let $s_0, s_1, \dots, s_k$ be the stores obtained by a sequence of derivation calls with pairwise fresh allocation parameters. Then for every $i \in \text{dom}(m_{s_0})$, $\llbracket i \rrbracket_{s_k}^- = \llbracket i \rrbracket_{s_0}^-$.*

Proof. Each step preserves ancestry closure and fixes ancestry behaviour on the identifiers present before it, by Theorem 5.7 and Theorem 5.12. Induction on k , noting that $\text{dom}(m_{s_0}) \subseteq \text{dom}(m_{s_j})$ for every j by Theorem 5.1. $\square$

The hypothesis matters. Theorem 6.1 covers derivation calls only, so a composite that also issues **save** under an existing identifier, or a deletion, is outside it; by Theorem 5.2 both of those can change an ancestry that a consumer has already read.

6.2 Memory composition

Before asking how memory composes with skills or with a harness, we ask how it composes with itself, since a harness that hosts several agents hosts several stores. The candidate operation is $\uplus$ of Theorem 3.5, and the question is whether operations are stable under enlarging the surrounding store.

Proposition 6.2 (Frame property for identifier-naming calls). *Let s, s' be stores with $\text{dom}(m_s) \cap \text{dom}(m_{s'}) = \emptyset$, and let a be a derivation call all of whose identifier arguments are exact identifiers in $\text{dom}(m_s)$, under an allocation parameter fresh for both. Suppose identifier resolution is by equality. Then $\alpha(s \uplus s')(a, \omega) = (o, t \uplus s')$ where $\alpha(s)(a, \omega) = (o, t)$.*

Proof. Under resolution by equality, each derivation method reads at most the records at the identifiers named in a , and those records are the same in s and $s \uplus s'$ because the domains are disjoint and $m_{s \uplus s'}$ extends m_s ; **supersede** reads none. The constructed records are a function of the retrieved records, the remaining arguments and ω , so they are the same in both cases, and the written edges are incident only to named identifiers and to the fresh ones. Hence the added records and edges are identical, the outputs are identical, and the resulting store is $t \uplus s'$. $\square$

Proposition 6.3 (The frame property fails for recall). *The hypothesis of resolution by equality is not satisfied by the implementation at a93035d, and without it Theorem 6.2 is false. `StorageRouter._recall_from_sqlite` resolves an identifier by the SQL predicate `id LIKE ?` with the*

pattern $f\{\text{memory_id}\}\%$ and takes $\text{cursor.fetchone}()$ from an unordered result set (Listing 4). Consequently there are stores s, s' with disjoint record domains and an identifier argument q such that $\text{out}(s, \text{recall}(q), \omega) \neq \text{out}(s \uplus s', \text{recall}(q), \omega)$.

Proof. Let s contain a single record at i and let q be a proper prefix of i . Then in s the predicate `id LIKE 'q%'` selects exactly the row at i , and the output is that record. Let s' contain a single record at $i'' \neq i$ with q also a prefix of i'' ; the domains are disjoint. In $s \uplus s'$ the predicate selects two rows. The query carries no `ORDER BY`, so the row returned by `fetchone` is determined by the resolution component π of the ambient parameter rather than by the query, and for a suitable π it is the row at i'' . The two outputs therefore need not agree. This is a second appearance of the same defect as Theorem 5.20: an operation whose result is decided by an enumeration order the interface does not mention. $\square$

The lineage observation fails to compose for a different reason, which is worth separating from the identifier question because it survives resolution by equality.

Proposition 6.4 (Extension can repair a dangling edge). *There are stores s, s' with disjoint record domains and an identifier $i \in \text{dom}(m_s)$ with $\llbracket i \rrbracket_s \neq \llbracket i \rrbracket_{s \uplus s'}$. Ancestry closure is therefore not preserved by restriction, and $\uplus$ does not preserve lineage behaviour on stores that are not ancestry-closed.*

Proof. By Theorem 5.30 a store with a dangling ancestry edge is reachable, so let s contain a record at i together with the edge $(i, \text{EVOLVED_FROM}, j)$ and no record at j . Then $\beta_s(j) = (\perp, \emptyset)$, so the behaviour $\llbracket i \rrbracket_s$ is the tree whose root carries $m_s(i)$ and whose single `EVOLVED_FROM` child carries $\perp$ and has no further children. Let s' consist of a single record at j ; the domains are disjoint. In $s \uplus s'$ we have $\beta(j) = (m_{s'}(j), \emptyset)$, so the same child now carries a record. The two behaviours differ in the first component of that child, hence $\llbracket i \rrbracket_s \neq \llbracket i \rrbracket_{s \uplus s'}$. If instead s is ancestry-closed then no such j exists, every $\mathcal{R}^-$ -edge from $\text{dom}(m_s)$ lands in $\text{dom}(m_s)$, and the argument of Theorem 5.7 applied to the inclusion shows $\llbracket i \rrbracket^-$ is unchanged. $\square$

Theorem 6.4 says that domain-disjoint union is the wrong composition operator for isolation: two tenants with disjoint record identifiers may still interfere, because one's dangling edge can acquire a target from the other. The repair is to strengthen the side condition, and with the stronger one the composition behaves.

Definition 6.5 (Isolated union). Write $\text{supp}(s) = \text{dom}(m_s) \cup \{i : (i, \rho, j) \in E_s \text{ or } (j, \rho, i) \in E_s\}$ for the set of identifiers a store mentions, in either component. For stores s, s' with $\text{supp}(s) \cap \text{supp}(s') = \emptyset$, the *isolated union* $s \uplus_! s'$ is $s \uplus s'$.

Proposition 6.6 (Isolated union preserves lineage behaviour). *Let $s \uplus_! s'$ be defined. Then $\beta_{s \uplus_! s'}(i) = \beta_s(i)$ for every $i \in \text{supp}(s)$, and every successor of such an i again lies in $\text{supp}(s)$. Consequently $\llbracket i \rrbracket_{s \uplus_! s'} = \llbracket i \rrbracket_s$ for every $i \in \text{supp}(s)$, and symmetrically for s' . The structure $(S, \uplus_!, \emptyset)$ is again a partial commutative monoid.*

Proof. Let $i \in \text{supp}(s)$. For the first component, $i \notin \text{dom}(m_{s'})$ since the supports are disjoint, so $m_{s \uplus_! s'}(i) = m_s(i)$. For the second, an edge of $E_{s'}$ with source i would put i in $\text{supp}(s')$, so the outgoing edges of i in $E_s \cup E_{s'}$ are exactly those in E_s ; and each of their targets lies in $\text{supp}(s)$ by the definition of the support. Hence β restricted to $\text{supp}(s)$ is unchanged and closed, and behaviour is determined by the restriction of the coalgebra to a closed subset. The monoid claim follows as in Theorem 3.6, since supp-disjointness of three stores pairwise is again symmetric in the three. $\square$

Remark 6.7 (Isolation is not what namespaces provide). Theorem 6.6 identifies the side condition a multi-tenant deployment needs: identifier supports must be disjoint, not merely record domains. Namespaces do not supply it at `a93035d`, for two independent reasons visible in the source. Identifier resolution does not consult the namespace: the predicate in Listing 4 filters on `id` and `deleted_at` only. And edge creation does not consult it either: `StorageRouter.add_edge` (`storage_router.py:1327`) compares no namespaces, so an edge between records of two namespaces is writable and puts each namespace’s identifiers in the other’s support. Isolation therefore has to be enforced above `ContextFS`, by partitioning identifiers, and not by the namespace field.

Remark 6.8 (The prefix regime is not narrow). It would be fair to object that prefix collisions are a measure-zero concern. They are not, for two reasons that follow from the identifier scheme rather than from any deployment. First, `Memory.id` defaults to `str(uuid.uuid4())[:12]` (`schemas.py:1371`). The string form of a version 4 UUID is grouped 8-4-4-4-12, so the first twelve characters are eight hexadecimal digits, a hyphen, and three more hexadecimal digits: eleven hexadecimal digits, that is 44 bits, all of them random, since the version nibble sits at string position 14. By the birthday bound, two identifiers coincide with probability about one half once the store holds roughly $1.18 \cdot 2^{22} \approx 4.9 \times 10^6$ records. Second, and more to the point, resolution is by prefix, and the documented interface invites abbreviation: the docstring of `ContextFS.recall` (`core.py:1288-1298`) says the identifier “can be partial, at least 8 chars”, and no code enforces the minimum. An eight-character query prefix fixes 32 bits, so two records match the same eight-character prefix with probability about one half once the store holds roughly $1.18 \cdot 2^{16} \approx 7.7 \times 10^4$ records. Ambiguity at the interface a caller is told to use therefore arrives some two orders of magnitude before ambiguity of the identifiers themselves.

The conclusion for composition is narrow and usable: memory stores compose by disjoint union for every operation that names full identifiers, and do not compose for the one operation that resolves abbreviations. A harness that shares one store across tenants must either resolve identifiers by equality or partition the identifier space, and namespaces do not partition it, because Listing 4 does not filter on `namespace_id`.

6.3 Composition with skills

Part II treats skills as operations of a coloured operad and takes the memory carrier as one of the colours a skill may consume and produce. What this Part supplies is Theorem 3.15, the type `Mem` of a handle on the carrier, and nothing else: in particular it does not supply a schema-indexed family of colours, because record schemas belong to the `Know` component in the sense fixed by Part IV rather than to the state space.

Two of the results above have direct consequences for a skill composite. A skill that reads the ancestry of a memory early in a composite and uses it late is sound by Theorem 6.1, provided no step of the composite deletes a record or saves under an existing identifier. A skill composite that is scoped to a namespace does not thereby scope its memory effects to that namespace, by Theorem 5.18 and Theorem 5.19: a merge issued inside the composite may name arguments in another namespace and will file the result in the namespace of its first argument. Scoping a composite is therefore not the same as scoping the state it touches, which is exactly the kind of gap the operadic reading of Part II cannot see, because operad substitution constrains the composition of operations and not the effects of the operations composed.

Part II does carry one mechanism that bears on this [10]. Its operations are graded by an effect class, and as stated there an operation whose input colour is `Mem` is not pure, so the grading determines where such an operation may be scheduled relative to its siblings. That grading orders memory-touching operations relative to one another. It does not bound which records they touch,

which is what Theorem 5.18 is about, so the two mechanisms are complementary rather than overlapping.

6.4 Composition with the harness

A harness surrounds a model with memory, tools, wiring and policy, and the Architecture triple (G, Know, Φ) is one categorical account of that layer [14, 1]. We use only the boundary fixed in Section 1.4: `Know` is a collection of checkable statements `cert`, and S is the coalgebraic state. Applied to `ContextFS`, the schema registry (`types/registry.py`, `class SchemaRegistry`, line 23) and the structured-data validation performed by `Memory`'s model validator (`schemas.py:1411`) are `Know`-level: they are checks on records rather than records. The record store and the edge set are S .

What Theorem 5.27 contributes to that discussion is a limit on what a `Know`-level check over memory can be: no check evaluated against a `ContextFS` store at `a93035d` can have the form “at transaction time τ_t the system believed P of the world at valid time τ_v ”, because the store does not distinguish the second parameter.

The boundary has a consequence on both sides. As stated in Part IV [12], the harness carries a projection of session content into its own store, which is state but is neither a certificate nor a memory coalgebra in the sense of Theorem 3.14: it has no derivation operations and no lineage edges. The same negative reading applies in the protocol layer, where, as stated in Part III [11], the append-only operation chain of `CatDB` is a wire-level ordering discipline and not a structure map $\alpha: S \rightarrow FS$. We agree with both readings. What makes a store a memory coalgebra on this account is not persistence and not an ordering on writes; it is the presence of derivation operations whose lineage is observable, which is what Theorem 3.7 asks for and what neither of those two stores has.

One cross-cutting dependency runs against the presentation order. A skill may call a memory operation at run time, so the companion treatment of skills depends on this one operationally, while nothing here depends formally on any companion. At the pinned commits this is a composition of models rather than of running code: `AgentHero` at `1c3ad24` declares no dependency on `ContextFS`, as reported in companion work on the harness [12].

7 Limits and counterexamples

Four laws fail in Section 5. Further limits concern the cost of repairing the temporal gap, the second unpersisted memory model in `ContextFS`, and the part of the system the coalgebraic reading does not reach at all.

7.1 Bitemporal repair

Theorem 5.27 is a statement about a record schema, not about an architecture, and the corresponding repair is small. We state it as a proposition about the model so that it is clear the lineage machinery survives unchanged.

Proposition 7.1 (The single-axis coalgebra is a quotient of a bitemporal one). *Let $\text{Rec}^{bt} = \text{Rec} \times \mathbb{T}$, with the second component read as a valid-time instant, and let A^{bt} extend the alphabet of Theorem 3.7 by giving `save`, `evolve`, `merge` and `split` one additional argument in $\mathbb{T}$. Let S^{bt} be the corresponding set of stores and α^{bt} the structure map that behaves as α on the `Rec` component and stores the extra argument in the $\mathbb{T}$ component of each constructed record. Let $u: S^{bt} \rightarrow S$ delete the second component of every record and let $\varepsilon: A^{bt} \rightarrow A$ delete the extra argument. Then for all $s \in S^{bt}$,*

$a \in A^{bt}$ and ω ,

$$\alpha(u(s))(\varepsilon(a), \omega) = (u_{\text{Out}}, u)(\alpha^{bt}(s)(a, \omega)),$$

where u_{Out} deletes the valid-time component of a record-valued output. The observation $\text{obs}^{bt}(s, \ell, \tau_v, \tau_t)$, defined by selecting the latest record with clock field at most τ_t among those with valid-time field at most τ_v , is not degenerate in valid time, while its image under u is.

Proof. The displayed equation is immediate from the construction: α^{bt} agrees with α on the Rec component by definition, and u deletes exactly the component that the extra argument feeds and that α never reads. Non-degeneracy of obs^{bt} is witnessed by the two histories of Theorem 5.27: recorded through A^{bt} they differ in the valid-time field of the second record, and obs^{bt} at (15 Feb, 15 Mar) returns the second record for one and the first for the other. Degeneracy of the image under u is Theorem 5.27(2). $\square$

Remark 7.2 (Cost of the repair). Theorem 7.1 touches neither $\mathcal{R}$ nor the lineage graph nor any of the laws L1 to L5 and L7. In implementation terms the repair is: one nullable time column on the **memories** table, one field on **Memory**, one optional argument on the four derivation methods and on the three agent-facing surfaces, and a comparison against that field in **Timeline.at**. It belongs here because the same theorem is cited by the synthesis of the series, that the gap is a design gap and not an obstruction: nothing about the coalgebraic reading of memory resists two axes, and the reason **ContextFS** has one is that its operations timestamp themselves rather than being told when their content became true.

7.2 Repository memory models

The analysis above concerns the persisted model: records in SQLite, edges in **memory_edges**, derivation through **MemoryLineage**. **ContextFS** also contains a second, richer memory model in **types/versioned.py**, and at **a93035d** the two do not meet.

The second model is the one that would satisfy more of Theorem 3.21. **VersionEntry** (line 87) carries a **reason** drawn from the four-valued **ChangeReason** and an **author**; **Timeline** (line 175) keeps entries sorted by timestamp and offers **at**, **since** and **by_reason**; **VersionedMem** (line 306) adds an authoritative-version pointer and a checkable invariant, **is_consistent** (line 576), which tests that timestamps are monotone, that all entries share a memory identifier, and that no version identifier repeats. That invariant is exactly the kind of internal consistency L2 and L3 ask for, and it is stated and checkable in the source.

It is, however, not reachable from the store. A **VersionedMem** is constructed on demand by **Memory.as_versioned** (**schemas.py**:1995) or **Mem.as_versioned** (**types/memory.py**:278), both of which call **VersionedMem.from_memory** (**types/versioned.py**:359); that constructor builds a timeline containing at most one entry, derived from the single record it is given, and only when that record has non-empty **structured_data** (lines 383–396). No code path in **storage_router.py** or **memory_lineage.py** at **a93035d** persists a **Timeline** or a **VersionEntry**, and the **memories** table (**core.py**:203–220) has no column for one. The alternative constructor **VersionedMem.from_lineage** (line 402) reconstructs a timeline from lineage history dictionaries, so the richer model is designed to be derived from the poorer one rather than to replace it.

The consequence for this paper is a scope statement. The results of Section 5 are about the persisted model, which is the one an agent reaches through the facade, the Model Context Protocol server and the command line. They are not claims about **types/versioned.py**, whose **is_consistent** may well hold of every timeline it constructs. The consequence for the system is a

question rather than a defect: the module that reasons about why a belief changed is the module whose output is not written down.

7.3 Missing run-scoped state

Remark 7.3 (There is no `RunContext`). The correspondence this series tests names `RunContext` as the concrete component playing the role of coalgebraic state, and `BiTemporalMemory` as the memory implementation [1]. Both are identifiers of that paper’s own reference implementation. Neither occurs in `ContextFS` at a93035d. The nearest structural analogue of a run-scoped state object is `Session` (`schemas.py:2035`), which carries a namespace, an originating tool, a device name, a repository path and branch, a list of `SessionMessage` entries (`schemas.py:2025`), start and end instants, a summary and a list of identifiers of memories created during the session; its lifecycle is `ContextFS.start_session` (`core.py:2061`), `end_session` (line 2115) and `load_session` (line 2185). That is a different object with a different field set under a different name, and this paper claims only that `ContextFS` independently realises an analogous structure. It does not claim that `ContextFS` implements the vocabulary of any other reference implementation.

7.4 Coalgebraic scope

The most useful limit to state is the one where the model has nothing to say. `ContextFS` is not only a lineage store; it is a retrieval system, and retrieval is how an agent actually uses it. `ContextFS.search` (`core.py:968`) and `StorageRouter.search` (`storage_router.py:435`) route queries to a full-text index and, when configured, to a vector store. Neither is an observation of (m_s, E_s) in the sense of Theorem 3.14, and the reason is visible in the write path. `StorageRouter.save` (line 89) commits the record to SQLite first, described in the source as authoritative, and then attempts the vector-store write inside `try: ... except Exception as e: logger.warning(...)` (lines 103–106), followed by an optional graph synchronisation under a second handler (lines 109–113). The vector index is therefore not determined by the record store along every path in the code, and even where it is, the result of a similarity query depends on an embedding function and on index parameters that are not part of the state.

One could enlarge the state space to include the index. That would be honest but not useful: the enlarged state would have no laws, because nothing in Theorem 3.21 constrains an approximate nearest-neighbour structure, and the enlarged coalgebra’s behaviours would distinguish states that no consumer can distinguish. The better description is that `ContextFS` has two faces. The lineage face is a coalgebra with checkable laws and one proved stability property. The retrieval face is a ranking function over the same records, and the vocabulary that fits it is information retrieval, not universal coalgebra. Reporting the correspondence for the first face and declining it for the second is the scoped claim of Section 1.1 applied honestly to this pillar.

Remark 7.4. This limit is different in kind from the five failed clauses. A failed clause is a defect that a patch could fix. Section 7.4 describes a part of the system for which the formalism is the wrong tool, and no patch to `ContextFS` would change that. When the series asks at which point the categorical account stops paying for itself, this is Part I’s answer.

8 Related work

Coalgebra for systems. The identification of state-based systems with coalgebras for a `Set`-endofunctor, and of behavioural equivalence with the kernel of the map into a final coalgebra, is Rutten’s [15]; Jacobs’s book is the standard extended treatment [5]. Our two functors are

unremarkable within that framework: $FX = (\text{Out} \times X)^{A^+}$ is a deterministic machine with input and output, and $HX = \text{Rec}_\perp \times \mathcal{P}_{\text{fin}}(\mathcal{R} \times X)$ is a finitely branching labelled transition system with state observations. What is specific here is the use to which the framework is put. We do not construct a final semantics for **ContextFS**; we use finality only in Theorem 5.12, and the substantive work is the identification of the sub-functor H^- for which the inclusion of existing identifiers is a homomorphism. Restricting a labelled transition system to a sub-alphabet in order to recover a homomorphism that fails on the full alphabet is a standard move; applying it to separate the immutable and mutable directions of a lineage graph appears not to have been done for agent memory.

Temporal databases. Valid time and transaction time as independent dimensions, and the definition of a bitemporal relation as one timestamped in exactly one of each, are due to the temporal database community; Jensen and Snodgrass give the consensus semantics [6] and Snodgrass’s book the practical treatment [17]. Theorem 5.27 is a negative result relative to that literature, and it is worth being clear about what is and is not new in it. That an append-only log with one clock cannot answer valid-time queries is not new. What the theorem adds is the precise form of the failure for a system whose designers describe it as history preserving: the recording map identifies histories that differ in valid time, so the deficiency is in the schema and not in the query implementation, and no improvement to **Timeline.at** could recover the distinction.

Immutability and merge. The design pattern of deriving new records rather than mutating old ones, and of treating the accumulated log as the system of record, is argued for at length by Helland [4]. **ContextFS**’s derivation operations follow it. The merge operation is closer to the conflict-free replicated data type literature [16]. The failure identified in Theorem 5.20 has a familiar form: the result depends on something other than the states being merged. The comparison has to be made at the right level, though. Asking for idempotency on the nose is a category error for an operation that allocates a fresh identifier on every call, and the correct question for a coalgebra is whether the resulting states are behaviourally equivalent. Theorem 5.24 settles that question positively: under an observation blind to allocation, merging the same arguments twice yields a store bisimilar to merging them once. Theorem 5.25 settles the other direction negatively, and identifies the two order-sensitive fields responsible. The difference from a replicated data type is therefore not that **ContextFS**’s merge fails the lattice laws outright, but that it satisfies one of them only after quotienting and fails the other by design, since it records provenance in argument order.

Typed agent memory. A prior paper by the present author describes a typed memory layer for an agent operating system in Elixir, with schema registries, causal versioning and a graph relationship manager [8]. That work and this one share a name for their reference implementation and little else: it is a design paper about a different runtime, it argues from the type system rather than from coalgebraic laws, and it makes no bitemporal claim. It is cited here as prior art by the same author, and its numbering is unrelated to this series: its Part III is a memory paper, this series’ Part III is about protocols.

The four pillars and the Architecture triple. Zhou et al. supply the pillar taxonomy [20]; de los Riscos, Corbacho and Arbib supply the Architecture triple [14]; Banu proposes the correspondence between them and validates it against one reference implementation [1]. Meng et al. give the alternative enumerative taxonomy of harness components [9], and Liu’s typed lambda calculus for agent composition is a different formal route to the same territory [7]. Cao’s argument for why

externalization is happening at all is economic and complexity-theoretic rather than structural [2]. This paper contributes one row of the correspondence, checked against a system that was not built with the correspondence in mind, and reports the row as partial; the remaining three rows and the question of where the correspondence stops paying for itself are taken up in companion work [10, 11, 12, 19].

Operads and wiring. Companion treatments of skills and protocols use the operad of wiring diagrams and its typed refinements [10, 11, 18, 3]. We mention the line only to place the boundary: the memory coalgebra composes with itself by the partial monoid of Theorem 3.6, and the operadic machinery those Parts need is not required here.

9 Open problems

Each problem has an explicit criterion for resolution.

1. *The second axis.* Theorem 7.1 shows a valid-time field is sufficient to make the observation non-degenerate. It does not show what the derivation operations should do with valid time when their arguments disagree. If **merge** receives three records with three valid-time instants, no choice among “earliest”, “latest” and “the interval spanned” is forced by the model, and the correct answer plausibly depends on the record type. Settled by: a merge rule for valid time whose induced observation satisfies a stated law, together with a proof that the law fails for the other candidate rules.
2. *A merge that reads only what it is given.* Theorem 5.20 identifies a tie-break that reads an ambient enumeration order. A one-line change repairs that merge path but not L5 globally, because prefix recall has a second order-dependent choice (Theorem 6.3). Deterministic resolution would be required as well. The merge change does not settle the algebraic question, which Theorem 5.24 and Theorem 5.25 leave in an odd state: idempotent up to allocation-blind bisimilarity, and not commutative, because the merged record records its sources in argument order. Whether a lineage merge should be commutative at all is genuinely unclear, since argument order is provenance. Settled by: either an argument-order-blind merge whose provenance is recovered from the edge set rather than from the content, together with a proof that it is commutative up to Theorem 5.23, or a proof that no such merge can distinguish histories the current one distinguishes.
3. *One ancestry representation.* Theorem 5.14 and Theorem 5.15 describe two representations, two fallbacks and two entry shapes. The question is not which to keep, since keeping the edges and deriving the metadata is the obvious answer, but what to do about stores already written under the current scheme, where the two representations may already disagree and there is no record of which is right. Settled by: a reconciliation procedure whose output is a store satisfying L2, together with a statement of what it does when the two representations conflict.
4. *Identifier resolution and composition.* Theorem 6.3 makes the frame property conditional on resolution by equality, and Theorem 6.8 gives the scale at which the condition starts to bite. The open question is whether a namespace-scoped resolution rule restores Theorem 6.2 in full, given that Theorem 5.18 shows namespaces are not respected by every operation. Settled by: either a proof that prefix resolution scoped to a namespace satisfies the frame property, or a counterexample of the same shape as Theorem 5.18.

5. *Persisting the version model.* Section 7.2 describes a module whose invariant `is_consistent` is exactly the internal-agreement property L2 and L3 approximate, and which is never written down. Whether the persisted lineage graph and a persisted timeline can be kept in agreement, and at what cost, is open. Settled by: a mapping from lineage graphs to timelines together with a proof that it takes stores satisfying L1 to L3 to timelines satisfying `is_consistent`.
6. *Deletion.* Every result in Section 5.2 carries the hypothesis that the operation is a derivation call, and Theorem 5.2 shows why. A memory system that must support erasure cannot be monotone, so the question is what weakening of Theorem 5.7 survives a deletion operation that is required to remove content while retaining the shape of the ancestry. Settled by: a coalgebra on stores-with-tombstones for which the analogue of Theorem 5.7 holds and under which erased records are not recoverable.

10 Conclusion

For ContextFS at commit a93035d, the coalgebraic model establishes that a record’s ancestry is invariant under later derivation on the ancestry-closed fragment, while its descendants may change. Five law clauses fail with named source witnesses, and the retrieval subsystem lies outside the useful scope of the coalgebraic account.

Three failures share one cause: merge typing, prefix resolution, and deletion consult an enumeration order that no interface declares. L5 exposes that hidden dependency directly. The strongest conclusion is therefore local: the model earns its cost where it converts implementation details into falsifiable stability and independence claims, and should not be extended to retrieval ranking merely to preserve a uniform vocabulary.

A Code evidence

Every code-backed claim in the body has a row below. Repository ContextFS is at `/Users/mlong/Documents/Development/contextfs-ai/contextfs`, commit a93035d; agent-os is at `/Users/mlong/Documents/Development/agentherowork/agent-os`, commit 61cb399. ContextFS paths in the third column are relative to `src/contextfs/`; the one agent-os path is relative to the repository root. Statements in the last column are implementation-level claims established by reading the file at the stated path and lines at the stated commit, not reports of runtime behaviour.

Claim	Repository, commit	Path	Identifier, lines	What the code shows
The relation set $\mathcal{R}$ is a closed enumeration of twenty-two labels (Section 4.2)	ContextFS a93035d	<code>storage_protocol.py</code>	<code>EdgeRelation</code> , 32–78	The implementation at a93035d declares an <code>Enum</code> with twenty-two members grouped as evolution, merge, split, reference, semantic, hierarchical, causal, resolution and dependency relations.
The inverse map is an involution with four fixed points (Theorem 4.1)	ContextFS a93035d	<code>storage_protocol.py</code>	<code>EdgeRelation.get_inverse</code> , 80–104	The implementation looks the label up in a twenty-entry table and returns the label itself on a missing key; the table pairs eighteen labels in nine mutually inverse pairs and fixes two.

Claim	Repository, commit	Path	Identifier, lines	What the code shows
The abstract carrier signature is save , recall , search and delete (Table 1)	ContextFS a93035d	storage_ protocol.py	StorageBackend, 148–249	The implementation declares a Protocol with save , save_batch , recall , search , delete and delete_by_namespace .
The abstract edge signature is separate from the record signature (Table 1)	ContextFS a93035d	storage_ protocol.py	GraphBackend, 347–547	The implementation declares a second Protocol carrying add_edge , get_edges , get_related , find_path and get_lineage .
The four derivation methods instantiate the derivation schema S1 to S4 (Theorem 5.6)	ContextFS a93035d	memory_ lineage.py	evolve 132–134, 152, 165, 170–180; merge 225–232, 256, 269, 275–285; split 397–399, 422, 436, 442–452; supersede 526–540	evolve , merge and split retrieve only their named identifiers and raise or skip when one is absent, construct records with fresh identifiers and a namespace copied from a retrieved record, and write ancestry-labelled edges out of the constructed identifier only. supersede retrieves and constructs nothing and writes two edges whose labels lie outside the ancestry vocabulary. No method removes an edge. S2’s freshness proviso is carried as a hypothesis and is not established by this code; see the next row.
Identifier freshness is sampled rather than checked (Theorem 5.3)	ContextFS a93035d	schemas.py, storage_ router.py	Memory.id 1371; StorageRouter. save 89; _save_to_sqlite 172	The identifier default factory is str(uuid.uuid4())[:12] ; no code path queries the store for that identifier before writing, and the insert statement is INSERT OR REPLACE , so a collision replaces the existing row without an error.
No derivation call mutates a retrieved record, and the three that construct records draw identifiers from the default factory (Theorem 5.1)	ContextFS a93035d	memory_ lineage.py	MemoryLineage, 61–91; evolve 93; merge 191; split 360; supersede 508	evolve , merge and split retrieve inputs through self._storage.recall , construct Memory instances with the default identifier factory, and persist them with self._storage.save at lines 165, 269 and 436; supersede neither retrieves nor constructs nor saves, writing only edges. No method assigns to a field of a retrieved object.
The ancestry edge for evolve originates at the constructed identifier (Theorem 5.7)	ContextFS a93035d	memory_ lineage.py	evolve , 170–180	The implementation writes EVOLVED_FROM from evolved.id to memory_id and EVOLVED_INT0 in the opposite direction.

Claim	Repository, commit	Path	Identifier, lines	What the code shows
The same holds for <code>merge</code> , <code>split</code> and <code>supersede</code> (Theorem 5.7)	ContextFS a93035d	<code>memory_lineage.py</code>	<code>merge</code> 275–285; <code>split</code> 442–452; <code>supersede</code> 526–540	Each pair is written with the derived record as source of the <code>_FROM</code> label and the original as source of the <code>_INTO</code> label; <code>supersede</code> uses <code>SUPERSEDES</code> and <code>SUPERSEDED_BY</code> , neither of which is an ancestry label. Whether the derived record’s identifier is fresh is the separate question of Theorem 5.3.
Ancestry is written twice, once guarded and once not (Theorem 5.14)	ContextFS a93035d	<code>memory_lineage.py</code>	<code>evolve</code> 158, 165, 168–182; <code>merge</code> 262, 273–287; <code>split</code> 426–432, 440–454	The implementation writes <code>evolved_from</code> , <code>merged_from</code> and <code>split_from</code> into the constructed record’s <code>metadata</code> and persists the record unguarded, then attempts the edge writes inside <code>try ... except Exception</code> handlers that log at warning level and do not re-raise.
Two lineage fallbacks disagree on the ancestor entry shape (Theorem 5.15)	ContextFS a93035d	<code>storage_router.py</code> , <code>memory_lineage.py</code>	<code>_get_lineage_from_sqlite</code> 1274; <code>get_lineage</code> 1683, 1712–1739; <code>get_history</code> 599, 639–663	The router emits ancestor dictionaries keyed <code>"id"</code> , <code>"relation"</code> , <code>"depth"</code> ; <code>get_history</code> emits dictionaries keyed <code>"memory_id"</code> and <code>"relation"</code> .
<code>link</code> writes an inverse only when asked (Theorem 5.16)	ContextFS a93035d	<code>memory_lineage.py</code>	<code>link</code> , 551, 580–587	The implementation writes the inverse edge inside <code>if bidirectional</code> : and obtains the label from <code>EdgeRelation.get_inverse</code> .
<code>merge</code> takes the namespace of its first retrieved argument unconditionally, and under the default union strategy unions metadata across all retrieved arguments (Theorem 5.18, Theorem 5.19)	ContextFS a93035d	<code>memory_lineage.py</code>	<code>merge</code> 225–230, 232, 256; <code>_apply_merge_strategy</code> 292, 298–307; <code>_generate_merged_content</code> 349–354	The retrieval loop skips identifiers that fail to resolve with a warning; the guard raises when fewer than two resolve; the constructor at line 256 sets <code>namespace_id=memories[0].namespace_id</code> outside any strategy branch, so it applies to every strategy; the union branch runs <code>combined_meta.update(m.metadata)</code> over every retrieved record, and the intersection, latest and oldest branches select metadata differently.
For a call that omits <code>memory_type</code> and uses the union, intersection or weighted strategy, the merged type is chosen by maximising a count over <code>set(types)</code> (Theorem 5.20)	ContextFS a93035d	<code>memory_lineage.py</code> , <code>schemas.py</code>	<code>_apply_merge_strategy</code> 307, 323, 345; <code>merge</code> 247–248; <code>MemoryType</code> 23	The implementation computes the type by maximising a count over a Python set of enumeration members in three of the five strategy branches; the caller’s optional <code>memory_type</code> argument overrides the result afterwards at lines 247–248; <code>MemoryType</code> is declared <code>class MemoryType(str, Enum)</code> .

Claim	Repository, commit	Path	Identifier, lines	What the code shows
No operation argument or record field carries a valid time (Theorem 5.27)	ContextFS a93035d	schemas.py, memory_lineage.py	Memory 1362–1398; evolve signature 93–101, metadata 159	Memory declares exactly two time-valued fields, <code>created_at</code> and <code>updated_at</code> , both defaulting to <code>datetime.now()</code> ; <code>evolve</code> 's parameters are identifier, content, summary, tag flags and a change reason, and its <code>"evolution_timestamp"</code> is assigned a UTC clock reading in ISO 8601 form.
The change reason is not exposed on any agent-facing surface (Theorem 5.29)	ContextFS a93035d	core.py, mcp/fastmcp_server.py, cli/memory.py	ContextFS.evolve 1515, 1545–1554; contextfs_evolve 623; evolve 247	The facade forwards five keyword arguments to <code>MemoryLineage.evolve</code> and <code>reason</code> is not among them; the Model Context Protocol tool and the command line both call the facade method.
Timeline.at compares against a single timestamp field (Theorem 5.28)	ContextFS a93035d	types/versioned.py	VersionEntry 87, 117; Timeline.at 238–263; from_memory 359, 383–396	<code>VersionEntry</code> declares one <code>timestamp</code> field; <code>at</code> returns the last entry whose timestamp is at or before the argument; <code>from_memory</code> sets that field from <code>memory.created_at</code> .
Edge endpoints are unchecked by default (Theorem 5.30)	ContextFS a93035d	storage_router.py	add_edge 1327–1370	The parameter <code>validate: bool = False</code> guards the endpoint existence query; every call site in <code>memory_lineage.py</code> omits the argument.
<code>save</code> can replace and <code>delete</code> removes edges in both directions (Theorem 5.2)	ContextFS a93035d	storage_router.py, core.py	<code>_save_to_sqlite</code> 172; ContextFS.save 673, 686; <code>delete</code> 586; <code>_delete_edges_for_memory</code> 626, 639	The insert statement is <code>INSERT OR REPLACE INTO memories</code> ; the facade accepts an explicit <code>id</code> ; deletion issues <code>DELETE FROM memory_edges WHERE from_id = ? OR to_id = ?</code> before deleting the record.
Recall resolves by identifier prefix without ordering (Theorem 6.3)	ContextFS a93035d	storage_router.py, core.py	<code>_recall_from_sqlite</code> 310–326; ContextFS.recall 1288–1298	The query is <code>SELECT * FROM memories WHERE id LIKE ? AND deleted_at IS NULL</code> with the pattern <code>f"{memory_id}%"</code> and no <code>ORDER BY</code> , followed by <code>fetchone</code> ; the docstring says the identifier can be partial with at least eight characters.
Identifiers carry forty-four random bits (Theorem 6.8)	ContextFS a93035d	schemas.py	Memory.id 1371	The default factory is <code>str(uuid.uuid4())[:12]</code> , whose first twelve characters of the grouped string form are eleven hexadecimal digits.

Claim	Repository, commit	Path	Identifier, lines	What the code shows
The versioned model is constructed on demand and never persisted (Section 7.2)	ContextFS a93035d	types/ versioned.py, schemas.py, types/ memory.py, core.py	VersionedMem 306; is_consistent 576; from_memory 359; from_lineage 402; Memory.as_ versioned 1995; Mem.as_versioned 278; memories DDL 203–220	Both entry points call <code>VersionedMem.from_memory</code> , which builds a timeline of at most one entry from a single record; the <code>memories</code> table declaration has no timeline column.
The schema layer is a checking layer, not part of the state (Section 6.4)	ContextFS a93035d	types/ registry.py, schemas.py	SchemaRegistry 23; Memory.validate_ structured_data_ schema 1411	The registry resolves schema types by name and the model validator rejects <code>structured_data</code> that fails the type’s JSON schema.
The nearest run-scoped object is <code>Session</code> , not <code>RunContext</code> (Theorem 7.3)	ContextFS a93035d	schemas.py, core.py	Session 2035; SessionMessage 2025; start_session 2061; end_session 2115; load_session 2185	<code>Session</code> declares identifier, label, namespace, tool, device, repository path and branch, messages, start and end instants, summary and created-memory identifiers.
The identifier <code>RunContext</code> does not occur in <code>ContextFS</code> (Theorem 7.3)	ContextFS a93035d	repository root	none	A recursive search of the working tree at <code>a93035d</code> for the literal string <code>RunContext</code> , excluding compiled caches, returns zero matches. This row is evidenced by a search over the repository rather than by a single file, and is the only row of this appendix of that kind.
The retrieval face is not determined by the record store (Section 7.4)	ContextFS a93035d	storage_ router.py, core.py	StorageRouter. save 89, 103–106, 109–113; StorageRouter. search 435; ContextFS.search 968	The implementation commits to SQLite first, described in the source as authoritative, and then attempts the vector-store and graph writes inside exception handlers that log and continue.
Prior typed memory work by the same author (Section 8)	agent-os 61cb399	papers/latex/ memory- layer.tex	abstract, 169–181	The abstract at commit <code>61cb399</code> describes a typed memory layer on Elixir with twenty-four memory types, a schema registry, vector-clock versioning, a graph relationship manager and a multi-backend storage router, and makes no bitemporal claim.

References

- [1] B. Banu. Harness engineering as categorical architecture. arXiv:2605.12239, 2026.
- [2] Z. Cao. The end of software engineering: how AI agents are fundamentally restructuring the software paradigm. arXiv:2606.05608v1, 2026. Retitled “Agentic software: how AI agents are restructuring the software paradigm” in a later version; citations here are to v1.

- [3] B. Fong and D. I. Spivak. Seven sketches in compositionality: an invitation to applied category theory. arXiv:1803.05316, 2018.
- [4] P. Helland. Immutability changes everything. *Communications of the ACM*, 59(1):64–70, 2016. DOI 10.1145/2844112.
- [5] B. Jacobs. *Introduction to Coalgebra: Towards Mathematics of States and Observation*. Cambridge Tracts in Theoretical Computer Science 59. Cambridge University Press, 2016. ISBN 9781107177895.
- [6] C. S. Jensen and R. T. Snodgrass. Semantics of time-varying information. *Information Systems*, 21(4):311–352, 1996. DOI 10.1016/0306-4379(96)00017-8.
- [7] Q. Liu. λ_A : a typed lambda calculus for LLM agent composition. arXiv:2604.11767, 2026.
- [8] M. Long. The AI operating system, Part III: memory layer, a typed filesystem for persistent agent cognition. YonedaAI Research Collective, 2026.
- [9] Q. Meng, Y. Wang, L. Chen, W. Wu, Y. Li, W. Jiang, Q. Wang, C. Lu, Y. Gao, Y. Wu, and Y. Hu. Agent harness for large language model agents: a survey. Preprints, 2026. DOI 10.20944/preprints202604.0428.v2.
- [10] M. Long. Operadic skill composition. Unpublished Part II manuscript in the Agentic Engineering repository, The YonedaAI Collaboration, 2026. <https://github.com/YonedaAI/agent-engineering>.
- [11] M. Long. Typed protocol wiring. Unpublished Part III manuscript in the Agentic Engineering repository, The YonedaAI Collaboration, 2026. <https://github.com/YonedaAI/agent-engineering>.
- [12] M. Long. Harness architecture. Unpublished Part IV manuscript in the Agentic Engineering repository, The YonedaAI Collaboration, 2026. <https://github.com/YonedaAI/agent-engineering>.
- [13] Python Software Foundation. The Python language reference: PYTHONHASHSEED and hash randomisation. <https://docs.python.org/3/using/cmdline.html#envvar-PYTHONHASHSEED>, accessed 2026.
- [14] P. de los Riscos, F. J. Corbacho, and M. A. Arbib. Working paper: towards a category-theoretic comparative framework for artificial general intelligence. arXiv:2603.28906, 2026.
- [15] J. J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249(1):3–80, 2000. DOI 10.1016/S0304-3975(00)00056-6.
- [16] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems (SSS 2011)*, LNCS 6976, pages 386–400. Springer, 2011. DOI 10.1007/978-3-642-24550-3_29.
- [17] R. T. Snodgrass. *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann, 2000.
- [18] D. I. Spivak. The operad of wiring diagrams: formalizing a graphical language for databases, recursion, and plug-and-play circuits. arXiv:1305.0297, 2013.

- [19] M. Long. The End of Software Engineering and the Rise of Agentic Engineering. Unpublished synthesis manuscript in the Agentic Engineering repository, The YonedaAI Collaboration, 2026. <https://github.com/YonedaAI/agent-engineering>.
- [20] C. Zhou, H. Chai, W. Chen, et al. Externalization in LLM agents: a unified review of memory, skills, protocols and harness engineering. arXiv:2604.08224, 2026.