

The End of Software Engineering and the Rise of Agentic Engineering

Synthesis

Matthew Long

The YonedaAI Collaboration, YonedaAI Research Collective

Chicago, IL

matthew@yonedaai.com

September 2026

Abstract

One account of agent systems explains why decision logic migrates from source code into a model that writes code at run time. Another proposes a formal description of the resulting harness of memory, skills, protocols and orchestration as an Architecture triple (G, Know, Φ) . This paper synthesizes a four-part test of the second account against three production systems and asks what it adds to the first. The pillar-to-triple interpretation works in both directions only under stated assumptions. A lax monoidal functor is constructed on two named harness fragments, not on the whole architecture category. Across four layers and three systems, conservative composition requires disjoint names; the skills implementation does not enforce that condition. We prove that the comparison map of a resource-indexed interpretation is invertible exactly when the factors have disjoint resource footprints. The criterion can therefore be checked before the composite exists. The formal account identifies which properties survive composition and which durable artifacts replace transient model-written code. It does not establish model behaviour, whole-category functoriality or interoperability among the three systems. Counterexamples include single-axis memory, accepted inputs with no supplier, policy information coupled to wiring, and different wiring fragments at the protocol and harness layers. The repository evidence is not publicly archived, so code claims require repository access; the mathematical results do not.

Contents

1	Introduction	4
1.1	Competing theses	4
1.2	Scope	5
1.3	Contributions	5
1.4	Evidence method	6
2	The interpretation	6
2.1	Bidirectional interpretation	6
2.2	Functor domain	8
3	Coalgebraic memory	9
3.1	Coalgebras and laws	9

3.2	Stable ancestry	10
3.3	Shared-store observations	10
4	Operadic skills	11
4.1	Candidate structures	11
4.2	Law audit	11
4.3	Composition gaps	12
5	Typed protocols	12
5.1	Protocol scope	12
5.2	Refusal algebra	12
5.3	Connection discipline	13
6	Harness architecture	13
6.1	Categories and preservation	13
6.2	Preservation and strength	14
6.3	Runtime responsibilities	14
7	Modular composition	15
7.1	The assembly	15
7.2	Named composition and the disjointness pattern	15
7.3	Nonconservative composition	17
7.4	Resource-footprint disjointness	18
7.5	Cyclic dependencies	21
7.6	Composition limits	22
8	Agentic engineering	22
8.1	Engineering question	22
8.2	Categorical contribution	22
8.3	Durable artifacts	23
8.4	Role of the triple	24
8.5	Scope	25
9	Limits and counterexamples	25
9.1	Single-axis memory	25
9.2	Open colour set	25
9.3	Coupled wiring and knowledge	26
9.4	Reference vocabulary	26
9.5	Combined limitations	27
9.6	Bibliographic corrections	27
9.7	Interpretive limits	28
9.8	Non-vacuity	28
10	Comparison with the compiler functors	29
10.1	Compiler checks	29
10.2	Modeled checks	29
10.3	Comparison limits	30
11	Open problems	30

11.1 Problems for the account	30
11.2 Required system changes	31
11.3 Falsification criteria	32
12 Conclusion	32
A Evidence map	32

1 Introduction

1.1 Competing theses

Cao argues that the arrival of capable language model agents is a restructuring of software engineering rather than an improvement of it [2]. The argument is stated from first principles. A traditional software system is a triple $S = (C, D, E)$ of computational resources, a set of deterministic decision rules encoded in source code, and an execution environment that evaluates the rules against inputs. Its defining property is that D does not change during execution. For a system with n components, each of which may or may not interact with each of the others, Cao’s Proposition 2.1 states the number of possible interaction paths as $P(n) \in \Theta(2^n)$. Human capacity to reason about those interactions he treats as essentially constant.

The stated rate is not supported by Cao’s derivation, which counts dependency graphs [2]. This series therefore corrects the count to $2^{\binom{n}{2}} = 2^{\Theta(n^2)}$ possibilities. The argument below uses only the weaker consequence that the quantity grows at least exponentially while human capacity to reason about it does not.

Cao defines an AI agent system as a tuple $A = (M, T, \text{Mem}, \Pi)$ of a model, a tool set, a memory subsystem and a planning mechanism, operating by $a_t \leftarrow M(s_t, \text{Mem})$ and $s_{t+1} \leftarrow \text{exec}(a_t)$. Its decision logic is generated at run time, and the code it emits, in Cao’s words, “is not the system; it is a transient artifact” [2]. He argues that complexity scaling motivates the agentic paradigm and that the delivery chain collapses from artificial intelligence through software to result into agent to result. He calls the resulting practice agentic engineering, with intent architects, agent coordinators and outcome auditors in place of code authors, using a definition from a multi-agent coordination account [7].

Cao’s empirical evidence is benchmark evidence: resolution rates on SWE-bench Verified [25] for an open development-process-centric model [14]. Those are set against a continuous-evolution benchmark on which scores fall from above eighty per cent on isolated tasks to at most thirty-eight per cent in continuous settings [3]. These measurements do not establish the counting argument or the proposed change in engineering roles.

Banu argues, independently and from a different direction, that the layer this migration produces already has a formal theory [1]. Zhou et al. organise the components that sit outside the model into four externalization pillars, memory, skills, protocols and harness [29]. Meng et al. give an alternative enumerative taxonomy of the same layer [17]. An industry account states the division of labour in the same terms, the model holding the capability and the harness making it useful [26]. The ArchAgents programme of de los Riscos, Corbacho and Arbib models an architecture as a triple $\mathcal{A} = (G_{\mathcal{A}}, \text{Know}_{\mathcal{A}}, \Phi_{\mathcal{A}})$ [20]. The syntactic wiring G is a graph of modules, ports and directed edges; the knowledge structure Know holds invariants and certificates; and the deployment map Φ sends abstract capability slots to concrete implementations. Its morphisms are the structure-preserving translations, which in practice are compilers. Banu maps the four pillars onto the triple row by row: Memory as state in a coalgebra, Skills as objects composed via an operad, Protocols as the syntactic wiring G , and Harness as the full triple. He reads an agent as a monoidal functor interpreting an architecture in a concrete system. His Definition 1 makes a certificate a triple $(\tau, \sigma, \text{evds})$ of a theorem statement, a map from theorem symbols to architecture parameters, and a derivation that can be mechanically replayed. His Definition 2 makes $\Phi: \text{Stages} \rightarrow \text{Models}$ a parameter of the architecture rather than a fixed constant, so that different deployments may use different Φ while preserving the same (G, Know) . Certificate preservation is imported from ArchAgents by citation and used operationally. A compiler claiming preservation must carry each source certificate’s theorem, parameters and replayable evidence into the target Know structure. Banu’s own compiler

checks certificate identity and verifier replay explicitly, because, as he says, functor laws alone are not sufficient. The evidence offered is five compiler functors, a two-model one-task escalation experiment, and a SWE-bench-lite run whose headline finding is a format-discipline ceiling at 8B parameters rather than a task-resolution gain.

The two arguments meet at a question neither asks. Cao says why the harness exists and does not say what it is; Banu says what it is and does not say why it should have appeared now. Between them sits an empirical question: whether the structure Banu describes is present in harnesses built by people who had never read his paper. Other formal routes into the same territory exist, among them a typed lambda calculus for agent composition whose survey of deployed frameworks reports widespread structural incompleteness [8]. The categorical route is taken here because its claims are stated about components a repository can be read for. Parts I to IV of this series answer that question for three production systems, one pillar at a time. This synthesis assembles the four answers and asks what the assembly is worth.

1.2 Scope

The series makes the following fixed and narrow claim, in the wording every Part uses.

The series does not claim the four pillars and the Architecture triple are “the same thing”. The claim is: under stated assumptions there is a structure-preserving interpretation of the four pillars in (G, Know, Φ) , and each of the three systems realises that interpretation on an identifiable fragment; the synthesis states where the interpretation fails (at least one counterexample). $\text{Memory} \rightarrow \text{Skills} \rightarrow \text{Protocols} \rightarrow \text{Harness}$ is the presentation order of the series; dependencies between the layers are stated explicitly, cycles included.

The correspondence is not exhaustively validated. Banu validates one reference implementation of his own and states its limitations; this series adds three independently developed systems and validates none of them against his implementation. None of the three systems implements Banu’s vocabulary. The six identifiers his four-pillar table names return no match in any of the three repositories at the pinned commits, as each Part records. Every imported result belongs to a Part, is cited as such, and carries that Part’s hypotheses; this paper proves two small results of its own and imports the rest. Failures of the interpretation are part of the claim, not exceptions to it.

The epistemic status of the whole is that of a case study on production systems that are not publicly archived. Part II records that **AgentHero** is a private repository which a reader cannot clone, and no Part supplies a public archive or a resolvable public identifier for any of the three. A claim about code here is therefore checkable by a reader with repository access and not by an anonymous one. That is weaker than an openly archived artifact, and nothing in this paper makes it equivalent. The mathematics does not depend on it: the definitions, theorems and proofs of Section 7, and those of each Part’s model section, stand whatever the repositories do next.

1.3 Contributions

The paper states the pillar-to-component interpretation in both directions and identifies the assumption each direction needs (Tables 1 and 2). It delimits the two harness fragments on which an interpretation has actually been constructed, then assembles the four layers without claiming that the three implementations interoperate.

Its new mathematical result is Theorem 7.7: a resource-indexed comparison map is invertible exactly when the factors have disjoint resource footprints. The corollary identifies strength with support exactness at the resource names rather than at the layer’s own names. The paper applies

that result to the harness, states the limits and counterexamples inherited from the four Parts, compares the conclusion with Banu’s compiler evidence, and maps every code-backed claim to its source. It introduces no new code claim or experiment.

1.4 Evidence method

The results proved here from definitions given here are Theorem 7.2, Theorem 7.5, Theorem 7.7 and Theorem 7.9, all in Section 7, and they are the paper’s own mathematical content. They are elementary, and that is deliberate rather than a concession. Their purpose is to be evaluated in four different settings by someone reading a schema or a manifest, so a criterion harder to evaluate than the question it answers would be of no use. Theorem 7.13 takes up the objection that elementary means empty.

Imported results are cited by their number in the Part that proves them, together with the label the Part gives them, so that a reader can find each statement, its hypotheses and its proof. Appendix A additionally gives each one’s source label, the identifier under which the corresponding evidence row appears in that Part’s own appendix. Nothing is restated as though it were proved here. Where an imported result carries a hypothesis that matters for the use made of it, the hypothesis is repeated.

Code claims use source inspection at a pinned commit and have the access limitation stated in Section 1.2. Every such claim belongs to a Part and was established there by reading a named file at a pinned commit. Each appears in Appendix A with the Part, the result, and the repository and commit. The four commits are `ContextFS a93035d`, `AgentHero 1c3ad24`, `CatDB 7cc9341a` and `agent-os 61cb399`. Throughout, “the implementation at commit X performs Y ” means that a reader of the named file at that commit will find Y written there. No source-backed property is described in the vocabulary of formal proof about a running system. Where a property is established by re-running a decidable check on retained data, it is described as checked by replay, which is the discipline Part IV fixes and this paper follows.

2 The interpretation

2.1 Bidirectional interpretation

An interpretation of the four pillars in the triple has two directions and they are not equivalent. The forward direction asks, for each pillar, which component of the triple it becomes and under what assumption. The backward direction asks, for each component, which pillar owns it and whether the pillars that touch it agree about what it is. Both directions were left implicit in the correspondence as first stated, and both turn out to need assumptions that are load-bearing.

The forward direction is given in Table 1. Each row names the pillar, the system that realises it in this series, the component, the definitions the responsible Part exports to make the row precise, and the assumption the row needs.

The assumptions are stated in full below, one per row.

Memory. The alphabet of the coalgebra must be extended by an ambient parameter absorbing allocation, the clock and the enumeration orders the implementation consults (Part I, Definition 3.8). Without that parameter the structure map is not a function at all. Part I argues that drawing the boundary narrowly would smuggle a negative result into a definition, so it absorbs every source of variation found. And the single-timestamp fields must be allowed to stand in for the bitemporal

Pillar, system	Component	Made precise by	Assumption, in brief
Memory, ContextFS	State S in a coalgebra ($S, \alpha: S \rightarrow FS$)	Part I, Definitions 3.4, 3.12, 3.14, 3.18, and the exported port type Definition 3.15	An ambient parameter, and a narrowing to one time axis
Skills, AgentHero	Operations of a coloured operad $\mathcal{O}$	Part II, Definitions 3.2, 3.7, 3.11	A colour for every port, and one output per node
Protocols, CatDB	Syntactic wiring G with typed ports	Part III, Definitions 3.1, 3.2, 3.3, 3.7, 3.10, 3.11, 3.12	The agent-facing surface, a thin type alphabet, and the tree fragment
Harness, AgentHero runtime	Full Architecture (G, Know, Φ)	Part IV, Definitions 3.1, 3.3, 3.8, 3.9, 3.17	Receipts count as certificates, names are sorted, and Φ is free

Table 1: The forward direction of the interpretation, pillar to component, with the definitions that make each row precise. The accompanying text gives the assumptions in full.

pair the series' notation reserves symbols for. That second assumption is stated as a narrowing rather than assumed silently, and it is proved to be one (Part I, Theorem 5.27).

Skills. A port that no tool schema mentions must be assigned the top colour if it is a value port and the artifact colour if it is an artifact port. Part II states this as Assumption 3.10 and calls it the idealisation the operad reading requires rather than a fact about the code. The reading of a manifest as an operadic composite must further be used only where it is faithful, which is for nodes declaring at most one output. No operation of an operad expresses the constraint that several outputs arise from one invocation (Part II, Theorem 7.2).

Protocols. The boxes and ports of G must be read off the agent-facing tool surface and not off the internal plan type alone. The type alphabet is the finite sets of field names ordered by reverse inclusion, so a port label is a set of names and carries no base type (Part III, Section 7.3). That is width subtyping for records and nothing more. And the fragment realised is the strict tree sub-operad rather than the full wiring operad (Part III, Section 7.4), because plan variants own their inputs and a subplan cannot be shared.

Harness. A receipt carrying a decidable identity check must count as a certificate. Part IV isolates that degenerate case as Definition 3.6 and bounds it with Proposition 3.7, which shows such a check is invariant under any relabelling of payloads that fixes the identity fields. Name values and payload values must be disjoint, the sorting hypothesis of Part IV, Remark 3.4, which the hook records satisfy and the manifest port vocabulary does not. And the deployment map must be a free parameter, as Banu's Definition 2 requires, which is what makes checks that read it unstable.

The backward direction is given in Table 2. Its first row is the one the series had to settle rather than assume, because two Parts use a wiring object and it was not obvious whether they use the same one.

Remark 2.1 (Two assumptions do most of the work). Two of the assumptions in Table 1 are load-bearing. The colour assumption for skills is an idealisation that Part II states rather than smuggles. Granting it entirely does not save the reading: even when every port is given a colour and all colours match, validation still accepts a manifest with an input port that nothing supplies (Part II, Example 5.11). The sorting hypothesis for the harness is satisfied by exactly the certificates

Component	Pillar that owns it	What the Parts settled, and the assumption it needs
G	Protocols, secondarily Skills	The series has two wiring objects, not one. As stated in Part III (Proposition 6.1), $\mathcal{W}$ and $\mathcal{O}$ are distinct operads with different concrete colour sets: any assignment from a CatDB port type to the manifest key at which a tool response is stored is constant on all port types stored at that key, so it is not injective, and nothing checks an assignment in the other direction because a manifest edge records node identifiers and no port datum. Part IV therefore fixes no single G for the series; it takes a general wiring in $\mathcal{W}_\top$ whose inner boxes are the stages of one installed application, which is not a tree wiring (Part IV, Remark 3.2)
Know	Harness, secondarily Protocols	Part IV owns the definition (Definition 3.3) and both other Parts cite it rather than restating it. The evidence is divided rather than duplicated: Part III reports that CatDB 's policy decisions and per-field provenance are certificate-shaped but not separately addressable, and that what one would call Know there is distributed over the port types and the reason sets (Part III, Remark 7.1); Part IV reports that AgentHero separates Know from G for the hook layer and not for the per-operation policy envelope, which is why its first law is satisfied only partially. Neither system separates them completely, and the assumption the direction needs is that a modelling choice about which part of G to call Know is admissible
Φ	Harness	A pair of name-keyed resolution functions, one for model providers and one for application process adapters, of the shape Banu's Definition 2 asks for. The assumption is Banu's own: that Φ is a parameter no morphism constrains. Part IV shows the exact price (Proposition 3.11): forgetting Φ is an equivalence of categories, so no functorial guarantee about deployment can be derived, and a check that reads Φ is reading data that morphisms may change
S	Memory	Mirrors the Memory row of Table 1. The one thing the other Parts need from it is the boundary: Know is what is checked, S is what is remembered, and a schema registry is on the Know side while a stored record is on the S side. Part I fixes that boundary itself so that nothing in Part I depends on Part IV

Table 2: The backward direction, component to pillar. The first row distinguishes the two wiring objects the series uses.

Part IV analyses and fails for the manifest port vocabulary. The structural result it supports therefore has a boundary running along the same line as Part II's. In both cases the assumption is not the obstacle; the missing check is.

2.2 Functor domain

Banu describes an agent as a monoidal functor interpreting an architecture in a concrete system. Part IV makes that precise as a lax monoidal functor $(A, \mu, \eta): (\mathbf{Arch}, \otimes, I) \rightarrow (\mathbf{Sys}, \boxtimes, J)$ (Definition 3.29). Here **Arch** is the category of architecture triples with the morphisms of Part IV, Definition 3.9 and the monoidal product of Part IV, Definition 3.17. The objects of **Sys** carry a set of named components labelled by realizers together with a set of receipt identities. They also carry an equivalence relation recording which of those identities a store design draws from a common monotone sequence (Part IV, Definition 3.24). The choice to record the partition into sequences rather than positions within a sequence is what makes a functor into **Sys** definable from a schema and never from a trace.

The functor is not constructed on all of **Arch**. Part IV constructs two interpretations on two full subcategories, and only these two exist in the series.

1. On **Arch_{db}**, the architectures whose certificates are durability-barrier certificates, Part IV defines A_{db} (Definition 6.1) and proves it lax monoidal and not strong (Theorem 6.3), under the hypothesis that the composite’s sequencing relation is total. That hypothesis is a claim about a database schema. Part IV discharges it for **AgentHero** at 1c3ad24 by exhibiting the barrier’s fence as a value derived from a **bigserial** primary key with no application scoping, so that every row of the table is co-sequenced with every other.
2. On **Arch_{hk}**, the architectures whose certificates are post-commit hook certificates, the receipt-sequencing abstraction is strong because fences are per row (Part IV, Proposition 6.7). It retains certificate scope as data but does not model the scope-based eligibility relation between certificates and events.

Part IV then shows that these two are the two sides of one criterion. Take an interpretation whose comparison map is bijective on components and on receipt identities. Strength holds exactly when no receipt identity from one factor is co-sequenced with a receipt identity from the other (Part IV, Theorem 6.5). Section 7.3 isolates the categorical content of that criterion and shows it is not special to harnesses.

The construction leaves several questions unresolved.

No interpretation of the whole of **Arch** into **Sys** is constructed anywhere in the series. The durability and hook fragments share a target and a definition schema, and it is plausible that they extend to a single functor on the subcategory generated by both. Nothing here proves that they do. A reader should not take “an agent is a lax monoidal functor” as an established fact about any system in this series. It is established on two named fragments.

The relation between the wiring object of the protocol layer and the wiring object of the harness layer is settled only negatively. Part III proves the erasure has no manifest-level section. Nothing constructs a comparison in the other direction, and whether a typed manifest edge would make one exist is open.

Whether the interpretation is functorial for the composition laws of the lower layers, that is, whether operadic substitution in $\mathcal{O}$ or in $\mathcal{W}$ is carried to anything in **Sys**, is not addressed. Part IV’s product $\otimes$ places no wires between the summands and models two harnesses installed side by side, not two harnesses connected to each other (Part IV, Remark 3.19). Connected composition is the subject of Parts II and III and has no harness-level counterpart in this series.

3 Coalgebraic memory

3.1 Coalgebras and laws

Part I [10] takes the first row of the correspondence, memory as a coalgebra (S, φ) for a polynomial functor. The row fixes neither the functor, nor the laws, nor a way of checking either against a system that was not built to satisfy them. Part I fixes all three, in the standard coalgebraic vocabulary [22, 5], for **ContextFS**, a memory layer for coding agents. That system stores typed records and exposes them over a command line, a Python facade and a Model Context Protocol [18] server. It also maintains a graph of relation-labelled edges recording how records were derived from one another.

A store is a pair $s = (m_s, E_s)$ of a finite partial map from identifiers to records and a finite set of relation-labelled edges (Definition 3.4). Two coalgebras are given over that data. The store coalgebra

(S, α) is for the functor $FX = (\text{Out} \times X)^{A^+}$, whose alphabet is the memory interface extended by an ambient parameter carrying fresh identifiers, the clock reading, and the enumeration orders the implementation consults (Definitions 3.12, 3.14). The extension is not optional, since without it α is not a function. The lineage coalgebra is for the finitary functor $HX = \text{Rec}_\perp \times \mathcal{P}_{\text{fin}}(\mathcal{R} \times X)$, whose final semantics is the ancestry of a record (Definition 3.18). Seven laws are stated so that each can be settled by reading source rather than by testing (Definition 3.21). The positive results are proved once for any realisation following a derivation schema (Definition 3.23), with the reading of the source entering at exactly one point.

3.2 Stable ancestry

The positive result is that ancestry is immutable. On an ancestry-closed store, and for any realisation following the derivation schema, the inclusion of the existing identifiers is a homomorphism of ancestry coalgebras, so no later derivation alters the ancestry of a record (Theorem 5.7). The descendant structure is not stable (Proposition 5.13). The guarantee is conditional, and the condition is characterised rather than assumed. Ancestry closure is an invariant of exactly the fragment built from the four derivation operations and from saves that allocate fresh identifiers (Proposition 5.9). It fails for deletion, which removes a record together with every edge incident to it in either direction, and for saves under an existing identifier (Proposition 5.2).

Five law clauses fail, each with a witness in the source. The merged record's type is settled by an enumeration order the interface never mentions (Proposition 5.20). Merge carries content and meta-data across a namespace boundary by construction rather than by leakage (Proposition 5.18), while operations naming only records of one namespace do have effects confined to it (Proposition 5.17). Ancestry is recorded twice and the two records can disagree (Proposition 5.14). Edge endpoints go unchecked by default (Proposition 5.30). And the stored state is single-axis.

The final failure concerns bitemporality, and it is proved rather than conceded. Bitemporality is taken in the sense of the temporal database literature, timestamps in exactly one valid-time and one transaction-time dimension [6, 23]. Take two bitemporal histories agreeing in every value, every transaction time and every change reason, and differing only in valid times. They have the same recording through the lineage interface, and their snapshot functions differ. So no function of the store recovers the distinction between a retroactive correction and a belated one, and every bitemporal observation is degenerate in valid time (Theorem 5.27). The quantifier ranges over all functions of the entire store, which is what makes it a separation result rather than a field count.

3.3 Shared-store observations

To a composite, the memory layer adds an observation that survives further activity. Along any sequence of derivation calls with fresh allocation parameters, the ancestry behaviour of an identifier present at the start is unchanged at the end (Corollary 6.1). A consumer that reads the ancestry of a record at one point in a composite is therefore entitled to assume it has not changed. It is not entitled to assume the same of descendants, and the guarantee does not survive deletion.

Memory composes with itself by union of stores, and the correct side condition is not the obvious one. Domain-disjoint union is a partial commutative monoid (Proposition 3.6) but does not preserve lineage behaviour, because one store's dangling ancestry edge can acquire a target from the other (Proposition 6.4). Requiring disjointness of the identifiers a store mentions in either component, not merely of its record domain, repairs this (Definition 6.5, Proposition 6.6). Namespaces do not supply that condition at the pinned commit, for two independent reasons visible in the source (Remark 6.7). The frame property for operations that name full identifiers holds (Proposition 6.2)

and fails for identifier resolution by prefix, which is the interface the documentation invites callers to use (Proposition 6.3).

Part I also records what the coalgebraic reading does not reach, which is the retrieval half of the same system. Similarity search over an external index is not an observation of the state space at all, because the write path commits to the authoritative store first and attempts the index writes inside exception handlers that log and continue.

4 Operadic skills

4.1 Candidate structures

Part II [11] takes the second row, skills as objects composed via an operad with serial, parallel and trace composition, and treats it as a hypothesis about **AgentHero**'s directed acyclic graph runtime. The word carries content [4]. A coloured operad has a fixed colour set, operations typed by input and output colours, a substitution operation, a symmetric group action and three axioms. If a runtime is to be called one, then somebody has to say what its colours are, what its operations are, what substitution means, and whether the axioms hold.

The colour set is defined explicitly (Definition 3.2) as a top element for an unconstrained value port, a family indexed by the schema expressions the manifest schema checker accepts, and one artifact colour, preordered by schema refinement. The skills operad is the free coloured symmetric operad on a signature with one generator per operation descriptor and declared output name (Definition 3.7). The descriptor retains handler and application identity as well as its port signature and node kind. A rooted manifest presentation with a distinguished output and a unique supplier map determines an operation by grafting (Definition 3.11); a general manifest need not determine one. Because the runtime's nodes have several outputs and an operad's operations have one, a coloured PROP is put beside the operad from the start (Definition 3.9) and both are carried through, with a table recording which results depend on the choice. Seven conditions are stated in two families that are deliberately not run together: the operad axioms transported to a realisation, and the conditions under which the operad reading is well posed at all.

4.2 Law audit

Three conditions hold. Composites are finite and well founded; the execution layering does not depend on how a plan was bracketed (Proposition 5.4); and concurrently scheduled operations are checked for interference (Code observation 5.6). One has a sufficient fragment and a sharper counterexample. Parallel composition is order-independent when sibling output key domains are disjoint (Code observation 5.16). A shared key with distinct values makes the last-write merge order-dependent (Example 5.17); a collision whose values agree does not.

Three fail. Validation accepts manifests with an input port that nothing supplies (Code observation 5.10, Corollary 5.12). The set of ports a node may read is fixed by the surrounding composite rather than by the node (Code observation 5.14). And the vocabulary has no unit (Code observation 5.20).

Theorem 7.2 separates correlation from nondeterminism. A **Set**-valued algebra can represent deterministic correlations among several outputs through their shared input, but cannot represent two output tuples for the same fixed input. A PROP preserves multi-output invocation syntax but remains deterministic. Proposition 7.4 separately shows that interchange can fail when actual output domains overlap. The order in which a run enters its handlers nevertheless has a well-defined class in the Mazurkiewicz trace monoid of the manifest's independence relation, for every validated

manifest and with no disjointness hypothesis (Proposition 5.8). The effect layer quotients correctly where the data layer does not.

4.3 Composition gaps

To a composite, the skills layer adds composability of work: a named unit with a declared interface that can be substituted into another unit’s input slot, together with a validation function that accepts or rejects the substitution before anything runs. The evidence supports part of that promise. The shape of a composite is checked and its independence structure is checked; its typing is checked only where a tool declares a schema.

The named combinators are derived rather than primitive. Gate and approval are guards, branch is an indexed family, map is bounded fan-out, and the loop construct is a bounded unrolling rather than a categorical trace (Code observation 6.4). The colour discipline the graph layer lacks is implemented one level up in the same repository, in the plugin capability resolver (Code observation 6.7), which turns the diagnosis into a concrete remedy. Part II is also explicit about which of its findings needed the formalism. That the runtime is dynamically typed is visible without it. The layer-wise disjointness condition, the multi-output obstruction, the trace-monoid soundness and the location of the missing check are not.

5 Typed protocols

5.1 Protocol scope

Part III [12] takes the third row, protocols as the syntactic wiring G with typed ports, and asks what a protocol is that a typed software interface is not. Every serialisation format has typed ports in a trivial sense, so a reading of the pillar that amounts to observing that requests and responses have types is a restatement rather than a theory.

The answer given is that a protocol is a typed box together with two further pieces of boundary data. The first is a refusal-complete outcome structure on each output port (Definition 3.10). It is a tuple of values, distinguished empty values, reasons, observable outcomes and an observation map. It is called refusal-complete when the observation map is injective on empty values and reasons together, so that a successful empty answer and each way of declining are distinct observable values. The second is a locus map (Definition 3.11) sending each reason to a name in a fixed ambient alphabet, so that a refusal raised inside a composite still names its origin when it reaches the boundary. A protocol is a box carrying both (Definition 3.12). All of it is boundary data on colours, so the operad’s separation of boundary from internals is preserved.

The wiring operad itself, in the line of Spivak and successors [24, 21, 27], is presented in the form the argument needs (Definition 3.3). Substitution resolves a demand by a walk that changes level and terminates in at most three steps, and the tree fragment is isolated as a sub-operad (Definition 3.7, Proposition 3.9).

5.2 Refusal algebra

The main structural result is that protocols form an algebra over the operad of strict tree wirings (Theorem 3.18), so refusals compose along substitution without being flattened, provided the composite takes a coproduct of inner reason sets rather than a quotient. The tree restriction cannot be dropped. With a multi-output box and sharing, composing in two steps gives a reason set a summand twice where composing the diagrams first gives it once, and the discrepancy is

over-reporting rather than a technicality (Remark 3.19). Part III also records that the engineering content is entirely in the hypothesis about coproducts. Mapping every inner reason to a string, or to a single upstream error variant, satisfies a type checker and destroys injectivity.

The main system-level result is a confinement theorem for CatDB’s query algebra (Theorem 5.2): in every plan the implementation accepts, the field names available at the root are contained in the union of the field names introduced at the leaves. The induction is elementary, and what it isolates is that two of the eight plan node cases carry the whole weight. The consequence for field-level authorization is a reduction and not a guarantee (Corollary 5.3). Part III records that the crate proving confinement cannot state the leaf hypothesis under which it would bear on authorization, because that crate does not depend on the policy crate (Remark 5.4).

Six conditions are checked against the source. Five hold on the stated fragments and encoding discrimination is not established (Table 2). Port compatibility holds for the field-name alphabet, which carries no base types. Leaf confinement holds with the authorization hypothesis discharged outside the crate. The wire format rejects unknown versions as serde errors, not as typed reasons belonging to the receiving box; the other transport uses path prefixes.

5.3 Connection discipline

To a composite, the protocol layer adds a discipline on the connections themselves, and specifically on what happens at a connection when no value can be produced. That addition is not redundant with either neighbour. The skills operad records that an operation has input colours and an output colour; it does not record how many ways the operation can decline, nor whether declining is distinguishable from producing nothing. A harness records checkable statements and replays their evidence; a check that fails to replay is a fact about the architecture, not a value that reaches the caller of a wire. Refusal is a value on a wire, and it is therefore the protocol layer’s object.

Four limits are recorded. The wiring and the policy and provenance layers are not separated at the crate level, and Part III is more precise than that. One fragment of the checkable material is compiled into the port types themselves, since the authorized field set is a leaf’s port label. The rest is carried as values in the reason sets, with no third place (Remark 7.1). The crate named for schemas is an eighteen-line placeholder at the pinned commit (Section 7.2). Port labels are field names and not value types (Section 7.3). A join bringing together two scans that both expose a field of the same name therefore produces one such field in the output set, and the disambiguation happens at the value level through per-field provenance rather than at the type level. And only the tree fragment is realised (Section 7.4), which Part III does not treat as a defect, since the confinement induction is over an owned term.

6 Harness architecture

6.1 Categories and preservation

Part IV [13] takes the fourth row and asks whether a harness built independently of the ArchAgents programme instantiates it. It defines the category **Arch** so that a certificate carries a claim identifier, a partial parameter map, a scope in the wiring and a decidable replay check (Definition 3.3). That is Banu’s Definition 1 with two changes forced by what implementations carry. The evidence component is split into a witness that an implementation retains and a checker that it runs, and a scope is added because real harnesses attach policy to individual stages as well as to whole systems. Morphisms preserve the claim identifier, transport the parameter map and the scope along the stage embedding, and require the target check to accept transported witnesses the source check accepted

(Definition 3.9). The monoidal product is disjoint union of wirings (Definition 3.17), and **Arch** is shown to be a symmetric monoidal category (Propositions 3.10, 3.18).

The target category **Sys** carries named components labelled by realizers together with receipt identities and a sequencing relation, and nothing a run could change (Definition 3.24). It is a product of two standard constructions rather than a transcription of one database schema (Proposition 3.26). An interpretation is a lax monoidal functor between them in the standard sense [16] (Definition 3.29). Certificate preservation is defined as replay stability, the strengthening of the morphism condition from an implication to an equivalence (Definition 3.30). That is a wide subcategory of **Arch** (Proposition 3.31) rather than an ad hoc predicate.

6.2 Preservation and strength

Part IV exports the following preservation and strength results.

The first is a separation of preservation (Theorem 3.32). A certificate whose check factors through (G, Know) and is receipt-shaped is replay-stable under every morphism; a certificate whose check reads Φ need not be, and the counterexample can be taken to be the identity on (G, Know) changing only the deployment. Both kinds occur in **AgentHero**, and Part IV names them. The price is Banu’s Definition 2: making Φ a free parameter is what allows one architecture to be deployed against different models, and it permits deployment-reading checks to become unstable. The theorem supplies a counterexample, not a universal statement about such checks. Part IV also records that certificate-wise preservation does not imply preservation of the derived admission predicate, which is antitone in Know (Proposition 3.34). A translation that preserves every certificate can still turn an admitted operation into a refused one, and installing an additional admission-phase hook is exactly such a translation.

The second is the laxness result (Theorem 6.3). On the durability fragment, the comparison map is a bijection on both underlying sets that is nonetheless not invertible in **Sys**, because the composite co-sequences receipts of independently installed applications and no morphism undoes that. Nothing is lost when two applications are composed; something is added, and structure that is added cannot be removed by a morphism (Remark 6.4).

The third is the criterion that makes the first two cases one (Theorem 6.5): for an interpretation whose comparison is bijective, strength holds exactly when the sequencing resource is local to a component. A design that wants strength must make its ordering resource local; a design that wants a single cross-application allocation order must give strength up. Calling that identifier a replay order requires additional consumer behaviour that the source evidence does not establish.

Seven laws are settled against the source (Table 2 of Part IV): five hold, one holds partially, and one fails. The failed law requires that no check read the deployment map, and the witness is an artifact-path containment test that reads the workspace root and an environment variable. The registry’s refusal to install two applications claiming the same type name is shown to be exactly the side condition that makes the monoidal product well defined on globally named architectures (Proposition 3.21).

6.3 Runtime responsibilities

Parts I to III each describe a component. None of them describes installing that component somewhere, deciding whether an operation on it is admitted, or recording that the operation happened, and those three are what the harness adds: they are Φ , the admission predicate, and the witness set. Each brings a failure mode the component layers do not have. Deployment brings the instability of checks that read it. Admission brings the antitonicity of the conjunction. Recording

brings laxness.

The harness’s certificates are receipt-shaped: they carry a schema string, an identity and a retained body, and no theorem. A receipt-shaped check is invariant under any relabelling of payload values fixing the identity fields, so it detects identity and record shape and is blind to content by construction (Proposition 3.7). Preservation of such a certificate means that a decidable identity check keeps accepting the same transported records. It does not mean that a property of the system is preserved, because no property is stated. The correspondence established is a correspondence of shape.

7 Modular composition

7.1 The assembly

The four layers assemble into one architecture (G, Know, Φ) as follows, with the qualifications each Part attaches.

The wiring G is a general element of the wiring operad $\mathcal{W}_T$ whose inner boxes are the stages of one installed application manifest (Part IV, Definition 3.1). It is not a tree wiring, because a manifest edge is a pair of endpoint lists, so one stage may supply several consumers, and a manifest may declare several terminal stages (Part IV, Remark 3.2). The protocol layer’s own wirings are tree wirings, so the two layers study different fragments of the same operad and a result proved by induction over a tree has no automatic counterpart at the harness level. Inside one application, the stages are skills, read as operations of $\mathcal{O}$ by grafting (Part II, Definition 3.11), and that reading is sound for the shape of G and not for a typed reading of its ports.

The knowledge layer Know is a finite set of certificates over G , presented equivalently as a stage-indexed family (Part IV, Remark 3.5) so that per-operation policy envelopes and whole-system hook subscriptions have one home. The evidence is divided between two systems and neither separates Know from G completely, as Table 2 records.

The deployment map Φ sends each stage to a realizer, and is realised by a pair of name-keyed resolution functions, one for model providers and one for application process adapters. It is a free parameter, and Section 6 records what that costs.

The state S is a memory coalgebra, and the harness relates to it in exactly one way that the series establishes. A skill’s port may carry the memory port type (Part I, Definition 3.15), so an operation of $\mathcal{O}$ may read and write memory. Such an operation is not pure in the effect grading, so the scheduler’s independence condition constrains where it may run (Part II, Definition 3.13).

At commit 1c3ad24 AgentHero composes one of the three. Its workspace manifest names neither ContextFS nor CatDB, and case-insensitive searches over the sources find no occurrence of either name (Part IV, Section 8.7). The skills runtime is in the same repository, so that composition is present in the artifact. The other two are compositions the model supports and the code has not made. The formal results are unaffected, since none of them mentions a particular component system, and the construction of Section 7.1 is a statement about the series’ model.

7.2 Named composition and the disjointness pattern

The four layers were developed separately across three systems. Conservative composition in each layer requires disjoint names, but the implementation status differs: the memory and harness operations are partial on that condition, the protocol model imposes it, and the skills executor accepts collisions and becomes order-sensitive. Part IV asks whether two of these cases are instances of one theorem. They share the following definition; Table 3 records the distinct implementation

status of each instance.

Definition 7.1 (Named composition system). A *named composition system* is a tuple

$$(\mathcal{C}, N, \text{supp}, \oplus, e).$$

Here $\mathcal{C}$ is a set of objects, N is a set of names, $\text{supp}: \mathcal{C} \rightarrow \mathcal{P}_{\text{fin}}(N)$ assigns to each object the finite set of names it mentions, $e \in \mathcal{C}$ has $\text{supp}(e) = \emptyset$, and $\oplus$ is a partial binary operation on $\mathcal{C}$ such that:

- (N1) $x \oplus y$ is defined whenever $\text{supp}(x) \cap \text{supp}(y) = \emptyset$;
- (N2) when $x \oplus y$ is defined, $\text{supp}(x \oplus y) = \text{supp}(x) \cup \text{supp}(y)$;
- (N3) $x \oplus y$ is defined if and only if $y \oplus x$ is defined, and their values are equal when defined;
- (N4) $e \oplus x = x = x \oplus e$ for every x ;
- (N5) $(x \oplus y) \oplus z = x \oplus (y \oplus z)$ whenever the supports of x , y and z are pairwise disjoint.

The system is *support-exact* when the converse of (N1) holds, that is, when $x \oplus y$ is defined only if the supports are disjoint.

Proposition 7.2 (Disjointness carries a partial monoid). *Let a named composition system be given, and denote it by $(\mathcal{C}, N, \text{supp}, \oplus, e)$. Write $\oplus_!$ for the restriction of $\oplus$ to pairs of disjoint support. Then $(\mathcal{C}, \oplus_!, e)$ is a partial commutative monoid: the operation is defined on every disjoint pair, it is commutative, e is a two-sided unit, and both bracketings of $x \oplus_! y \oplus_! z$ are defined and equal whenever the three supports are pairwise disjoint. The iterated composite of a finite family with pairwise disjoint supports is likewise defined, and is independent of the order in which it is formed.*

Proof. Definedness on disjoint pairs is (N1), commutativity is (N3), and the unit law is (N4) together with $\text{supp}(e) = \emptyset$, which makes e disjoint from everything. For the associativity clause, let x, y, z have pairwise disjoint supports. Then $x \oplus y$ is defined by (N1) and $\text{supp}(x \oplus y) = \text{supp}(x) \cup \text{supp}(y)$ by (N2), which is disjoint from $\text{supp}(z)$, so $(x \oplus y) \oplus z$ is defined by (N1); symmetrically for the other bracketing; and the two are equal by (N5). The last claim follows by induction on k : any two ways of forming the iterated composite differ by a finite sequence of applications of (N3) and (N5), each of which is available because every intermediate support is a union of some of the $\text{supp}(x_i)$ by (N2), hence disjoint from the supports of the remaining ones. $\square$

Theorem 7.2 is the same statement in four places, and the four places disagree about which name set to use, which is where the engineering content sits. Table 3 records the four instances.

Remark 7.3 (The four name sets are not the same name set). It would be an overstatement to say the four rows of Table 3 are one condition. They are four instances of one shape, and the instances are about different things: record identifiers, port keys within a manifest, supply ports within a diagram, and type identifiers across applications. Part IV states this correctly when it says the two conditions it compares are not the same condition but are instances of one pattern. What Theorem 7.1 adds is that the pattern has a definition, so that a designer at any layer knows which question to ask. What does an object mention, and is the operation defined only when two objects mention nothing in common? At three of the four layers the answer is available in the source; at the skills layer it is not, and that is the content of Part II's second open problem.

Layer	Objects and composition	Name set and support	Status, and the result that settles it
Memory	Stores under union	Identifiers; the support is every identifier the store mentions in either component, not the record domain alone	Support-exact once the correct support is chosen. Domain-disjoint union is a partial commutative monoid but does not preserve lineage behaviour (Part I, Propositions 3.6, 6.4); isolated union does (Part I, Definition 6.5, Proposition 6.6). Namespaces do not supply the condition (Part I, Remark 6.7)
Skills	Sibling nodes placed in one execution layer	Declared output port keys of the siblings	The composition is total in the implementation. Disjoint actual output supports suffice for order independence; a shared key carrying distinct values is a counterexample because the store merge keeps the last write (Part II, Code observation 5.16, Example 5.17). Equal-value collisions remain order-independent. The disjointness hypothesis governs the interchange law in the richer structure (Part II, Proposition 7.4)
Protocols	Boxes wired by a diagram	Supply ports, and separately the pinned source of a scan	Linearity is imposed rather than checked at run time: a tree wiring requires that no supply port is sent to by two distinct demands (Part III, Definition 3.7), and the implementation refuses a union whose inputs pin the same source (Part III, Proposition 5.5). Where sharing is allowed the algebra loses associativity (Part III, Remark 3.19)
Harness	Globally named architectures under the monoidal product	Global stage and type names	Support-exact, and the decision procedure is in the source. The untagged product is defined exactly on pairs whose name images are disjoint, decidable in $O(n \log n)$ with an ordered map, and that is exactly the check the registry performs before admitting a pair of installed applications (Part IV, Definition 3.20, Proposition 3.21)

Table 3: One side-condition shape across four layers. The final column distinguishes conditions imposed by the model, enforced by the implementation, and required for deterministic behaviour.

7.3 Nonconservative composition

The second thing the four layers share is less comfortable. In each of them there is an explicit witness that composing two objects produces something not determined by the two objects. Two mechanisms are involved and they are kept apart here, because only one of them has a categorical form.

The first mechanism is *addition*: the composite carries structure that neither factor had, and no map recovers the factors.

- At the memory layer, adjoining a second store to a store with a dangling ancestry edge supplies a record where the first store had none. The lineage behaviour of an identifier present in the first store therefore changes (Part I, Proposition 6.4). The observation at the dangling target was $\perp$ and is now a record, and no restriction recovers the earlier behaviour.
- At the harness layer, installing two applications into one deployment co-sequences receipt identities that were in separate sequences. The comparison map is a bijection on both underlying sets and is not invertible in the target category, because the added relation cannot be removed by a morphism (Part IV, Theorem 6.3, Remark 6.4).

The second mechanism is *under-determination*: the composite is defined but its value is not fixed by the factors alone.

- At the skills layer, two sibling nodes writing the same output key give a result decided by the order in which the stores are merged. The merge is two extend operations that overwrite on a repeated key (Part II, Code observation 5.16). The composite is not a function of the two operations.
- At the protocol layer, composing in two steps rather than one duplicates a reason summand when a shared inner box is upstream of two outputs of a multi-output box. The two composites differ, and the two-step one reports the same inner refusal twice (Part III, Remark 3.19).

Disjoint supports prevent the listed nonconservative behaviour in each case. Only some implementations enforce that condition, as Table 3 records.

The categorical content is confined to the first mechanism. The rest of this section isolates it, and the isolation does more than name a pattern. It converts a condition on a composite into a condition on the two factors, and it identifies the support function of Theorem 7.1 that a layer ought to be using.

7.4 Resource-footprint disjointness

We work in the following setting, which is the shape of the receipt component of the target category Part IV uses (Definition 3.24) and of nothing more.

Definition 7.4 (Relational category over B). Fix a set B . Let $\mathcal{D}_B$ be the category whose objects are triples $(U, \sim, g)$ of a set, a binary relation on it, and a function $g: U \rightarrow B$, and whose morphisms $(U, \sim, g) \rightarrow (U', \sim', g')$ are the functions $h: U \rightarrow U'$ with $g' \circ h = g$ and $u \sim v \Rightarrow h(u) \sim' h(v)$, composed as functions.

Lemma 7.5 (Reflection criterion). *Let $h: (U, \sim, g) \rightarrow (U', \sim', g')$ be a morphism of $\mathcal{D}_B$ whose underlying function is a bijection. Then h is an isomorphism in $\mathcal{D}_B$ if and only if h reflects the relation, that is, $h(u) \sim' h(v)$ implies $u \sim v$.*

Proof. Suppose h is an isomorphism with inverse k . Then k is the set-theoretic inverse of h , since kh and hk are identity functions, and k is a morphism, so it preserves the relation. If $h(u) \sim' h(v)$ then applying k gives $u \sim v$, so h reflects.

Conversely suppose h reflects and let $k = h^{-1}$ as a function. It commutes with the structure maps: from $g' \circ h = g$ we get $g \circ k = g' \circ h \circ k = g'$. It preserves the relation: if $v_1 \sim' v_2$, write $v_i = h(u_i)$ using surjectivity; reflection gives $u_1 \sim u_2$, that is, $k(v_1) \sim k(v_2)$. So k is a morphism and is two-sided inverse to h . $\square$

Theorem 7.5 on its own is a restatement of what an isomorphism is. It becomes informative when the relation is presented as the kernel of an assignment of resources, because the reflection condition then becomes a disjointness condition on the factors rather than a condition on the composite. That presentation is available in every case the series examines. A sequencing relation records which receipts are drawn from a common monotone sequence, so it is the kernel of the map sending a receipt to the sequence it is drawn from.

Definition 7.6 (Resource-indexed interpretation). Let $(\mathcal{C}, N, \text{supp}, \oplus, e)$ be a named composition system, and fix sets K of *resource names* and B of *observable fields*. A *resource-indexed interpretation*

assigns to each $x \in \mathcal{C}$ a set $A(x)$ together with maps $\kappa_x: A(x) \rightarrow K$ and $g_x: A(x) \rightarrow B$. To each pair x, y for which $x \oplus y$ is defined it assigns a bijection

$$\mu_{x,y}: A(x) \sqcup A(y) \longrightarrow A(x \oplus y) \quad \text{with} \quad \kappa_{x \oplus y} \circ \mu_{x,y} = [\kappa_x, \kappa_y], \quad g_{x \oplus y} \circ \mu_{x,y} = [g_x, g_y],$$

where square brackets denote copairing out of the disjoint union. Fix also the two injections $\iota_1, \iota_2: K \rightarrow K \sqcup K$ and the fold $\nabla = [\text{id}_K, \text{id}_K]: K \sqcup K \rightarrow K$, the identity on each summand and the unique map with $\nabla \iota_1 = \nabla \iota_2 = \text{id}_K$. The condition $\kappa_{x \oplus y} \circ \mu_{x,y} = [\kappa_x, \kappa_y]$ is then equivalent to $\kappa_{x \oplus y} \circ \mu_{x,y} = \nabla \circ [\iota_1 \kappa_x, \iota_2 \kappa_y]$. Each $A(x)$ is regarded as an object of $\mathcal{D}_B$ by taking $\sim_x$ to be the kernel of κ_x , that is, $u \sim_x v$ if and only if $\kappa_x(u) = \kappa_x(v)$. The sum $A(x) \sqcup A(y)$ carries the disjoint sum of the two kernels, which relates nothing across the summands. We require $A(e) = \emptyset$. The *resource footprint* of x is $\rho(x) = \kappa_x(A(x)) \subseteq K$.

The two conditions on $\mu_{x,y}$ say that composing does not move a receipt to a different resource and does not change what a receipt reports, which is what makes $\mu_{x,y}$ a comparison map rather than an arbitrary relabelling. The map $\mu_{x,y}$ is automatically a morphism of $\mathcal{D}_B$. It preserves g by hypothesis, and if $u \sim v$ in the sum then u and v lie in one summand with equal κ , so their images have equal $\kappa_{x \oplus y}$.

Theorem 7.7 (Strength is resource disjointness). *Let A be a resource-indexed interpretation and let $x \oplus y$ be defined. Then*

$$\mu_{x,y} \text{ is an isomorphism in } \mathcal{D}_B \iff \rho(x) \cap \rho(y) = \emptyset.$$

In addition, $\rho(x \oplus y) = \rho(x) \cup \rho(y)$ always.

Proof. Write $c = [\iota_1 \kappa_x, \iota_2 \kappa_y]: A(x) \sqcup A(y) \rightarrow K \sqcup K$ for the classifying map of the sum, where ι_1, ι_2 are the two injections of $K \sqcup K$, and write $\nabla: K \sqcup K \rightarrow K$ for the fold, the map that is the identity on each summand. The relation of $A(x) \boxtimes A(y)$ is $\ker c$, since two elements have equal c -value exactly when they lie in one summand with equal κ -value. The defining condition on $\mu = \mu_{x,y}$ reads $\kappa_{x \oplus y} \circ \mu = [\kappa_x, \kappa_y] = \nabla \circ c$, so the relation of the composite pulls back along μ to $\ker(\nabla \circ c)$.

Now $\ker c \subseteq \ker(\nabla \circ c)$ for any ∇ , which is the statement that μ preserves the relation, and by Theorem 7.5 μ is an isomorphism if and only if the inclusion is an equality. That happens if and only if ∇ is injective on the image of c , which is $\iota_1 \rho(x) \cup \iota_2 \rho(y)$. The fold identifies $\iota_1 k$ with $\iota_2 k$ for each $k \in K$ and identifies nothing else, so it is injective on that set if and only if no k lies in both $\rho(x)$ and $\rho(y)$.

For the last claim, μ is surjective and $\kappa_{x \oplus y} \circ \mu = \nabla \circ c$ has image $\rho(x) \cup \rho(y)$. □

Remark 7.8 (The comparison is change of base along the fold). The proof identifies what the comparison map is. The disjoint sum is classified by a map into $K \sqcup K$, one copy of the resource names per factor. The composite is classified by the same map followed by the fold $K \sqcup K \rightarrow K$, one copy for both. And μ is the map these two classifications induce. Laxness is therefore not a peculiarity of one database schema. It is what happens whenever a realization allocates one pool of ordering resources to a composite where the abstract product allocates one pool per factor, and the fold is the formal record of that allocation decision. The design question “one sequence or many” is the question of which of the two classifications the implementation performs.

Corollary 7.9 (Strength on a family is resource disjointness). *Let $\mathcal{F} \subseteq \mathcal{C}$ be closed under $\oplus$ where defined and contain e , and assume $A(x)$ is finite for every $x \in \mathcal{F}$. Then $\rho(e) = \emptyset$ and $\rho(x \oplus y) = \rho(x) \cup \rho(y)$ whenever the composite is defined. The comparison maps of A are isomorphisms for every composable pair in $\mathcal{F}$ if and only if every such pair has disjoint resource footprints.*

Proof. $\rho(e) = \emptyset$ because $A(e) = \emptyset$ by Theorem 7.6, and the union equation is the last assertion of Theorem 7.7. That theorem also says that a given composable pair has an invertible comparison exactly when its resource footprints are disjoint. Quantifying over the composable pairs in $\mathcal{F}$ gives the equivalence. $\square$

Theorem 7.9 carries the synthesis. A layer has two name sets in play: the one its composition law is partial over, and the one its realization draws its ordering resources from. Admissibility of composition is disjointness at the first; strength of the interpretation is disjointness at the second. These are separate conditions evaluated at two different name sets; a layer can satisfy either without the other. In particular, the corollary does not assert that ρ satisfies the associativity clause (N5), whose hypothesis is stated using supp .

Corollary 7.10 (The harness results are the two cases). *Take $\mathcal{C}$ to be the globally named architectures with $\oplus = \otimes_N$ and supp the image of the global name map, so that supp -exactness is Part IV’s admissibility result (Proposition 3.21). Then:*

1. *For the durability interpretation (Definition 6.1), K is the set of fence sequences and κ sends a receipt identity to the sequence its fence is drawn from. Then $\rho(\mathcal{A})$ is the one-element set naming the barrier sequence whenever $\text{St}(G_{\mathcal{A}})$ is nonempty. Two nonempty architectures therefore have intersecting resource footprints, and Theorem 7.7 gives that the comparison is not an isomorphism, which is Part IV’s laxness result (Theorem 6.3).*
2. *For the hook-fence abstraction, fix the canonical event-identifier set E from Part IV, Proposition 6.7, and an ambient set $\text{Cert}_{\text{glob}}$ of globally tagged certificate identities. For each architecture let $j_{\mathcal{A}}: \text{Know}_{\mathcal{A}} \hookrightarrow \text{Cert}_{\text{glob}}$ be the canonical inclusion, chosen compatibly with the tensor tags: $j_{\mathcal{A} \otimes \mathcal{B} \iota_1} = j_{\mathcal{A}}$ and $j_{\mathcal{A} \otimes \mathcal{B} \iota_2} = j_{\mathcal{B}}$. Take $K = \text{Cert}_{\text{glob}} \times E$ and $\kappa_{\mathcal{A}}(c, e, i) = (j_{\mathcal{A}}(c), e)$. The factor and composite maps therefore use the same global resource names; tensor tagging changes only their local presentation. The images of the two factors are disjoint, so Theorem 7.7 gives the receipt-sequencing result of Part IV’s Proposition 6.7. That proposition does not model the scope-based eligibility relation between a certificate and an event.*

Part IV’s criterion (Theorem 6.5) is the special case of Theorem 7.7 in which the resource footprint is not named, so that the condition is read off the composite rather than off the factors.

Remark 7.11 (What the resource form buys over the criterion it generalises). Two consequences follow. The condition moves to the factors. Part IV’s criterion asks whether two receipt identities are co-sequenced in the composite, which is a question about an object that does not exist until the composition is performed. Theorem 7.7 asks whether two sets computed from the factors alone are disjoint, which is answerable before installing anything. And the condition acquires a name that the composition law already had. Once the resource footprint is written down it is a support function in the sense of Theorem 7.1. The question “is this interpretation strong” then becomes an instance of the question every layer of this series was already asking. Theorem 7.9 says the two questions have the same shape rather than merely resembling one another.

Remark 7.12 (The memory layer chose the resource footprint by hand). The cross-layer observation Theorem 7.9 makes available is this. At the memory layer the ordering resource is the identifier space: what one store can disturb in another is exactly what the two of them both mention. Part I arrived at that by inspection, finding that composition on disjoint record domains fails to preserve lineage behaviour and that composition on disjoint identifier supports does (Part I, Proposition 6.4, Definition 6.5 and Proposition 6.6). The correct support was the set of identifiers a store mentions in either component rather than the set it stores. In the vocabulary of Theorem 7.6 the second is

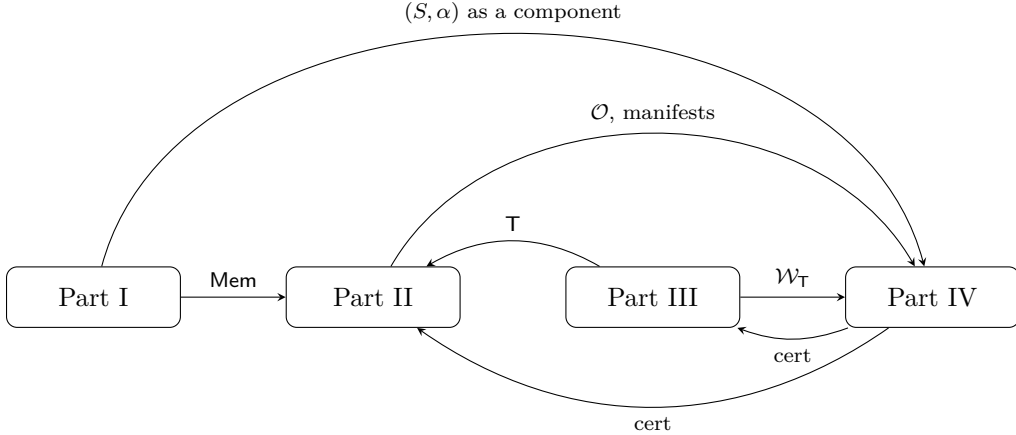


Figure 1: The definitional import graph. An arrow from P to Q means Q uses an object exported by P . Three arrows run against the presentation order left to right: the port-type alphabet flows from Part III to Part II, and the certificate notion flows from Part IV to Parts II and III.

the resource footprint and the first is not. Part I’s repair is therefore the passage from a support that ignores the shared resource to one that tracks it, which is the move Theorem 7.7 says is forced.

Two qualifications keep this an analogy rather than a second instance of the theorem. Part I’s target is not an object of $\mathcal{D}_B$: what changes there is the value of a lineage observation, not the extent of a relation, so Theorem 7.7 does not apply to it as stated. And the two under-determination cases of Section 7.3 are not of this kind at all, since there the composite is defined and merely not a function of the factors. The theorem covers one of the four layers and names the pattern the other three exhibit; saying which is which is the difference between a pattern and an equivocation.

Remark 7.13 (On the objection that this is a restatement). Read at the level of the one system on which it was discovered, the laxness result reduces to the remark that rows sharing one database sequence interleave, and Part IV notes that reduction explicitly. Theorem 7.7 is an equivalence: under its hypotheses, a shared sequence is the only obstruction, and changing it suffices. Its condition is on the factors, so it is decidable before the composite exists. The formulation also applies where the concrete answer is not already known. Nobody looking at a workflow runtime would think to ask whether two sibling nodes draw on a shared write-ordering resource. The answer at that layer, as stated in Part II, Code observation 5.16, is that they do, and that the resource is the output key namespace of the store.

7.5 Cyclic dependencies

Memory, Skills, Protocols, Harness is the order in which the series presents the pillars. It is not the order of logical dependency, and three edges of the actual graph run against it.

Part I imports nothing formal. It needs **Set**, the definition of a coalgebra and its homomorphisms, and the bitemporal vocabulary of the temporal database literature. It exports the memory port type to Part II and its separation theorem to Part IV.

Part II imports the port-type alphabet T from Part III, which is presented after it, and the memory port type from Part I. It cites Part IV for the certificate notion when it observes that a per-tool policy envelope is stage-indexed knowledge. None of these imports is load-bearing for a proof in Part II: the only property of T used there is that it is a set of colours, and the concrete alphabet analysed is defined in full in Part II itself.

Part III imports the colour set of Part II for its type-erasure statement and the certificate notion from Part IV, and exports the wiring operad, its tree sub-operad and the type-erasure result to Part IV.

Part IV imports from all three and exports nothing to them that is used in a proof. Restricted to formal dependencies the graph is acyclic with Part IV as its unique sink, which is what Part IV states (Section 7.5).

The cycles the series has to state are operational, not definitional, and there are two. A skill may call a memory operation at run time, so an operation of $\mathcal{O}$ may have a port carrying the memory carrier, while nothing in the memory layer depends on the skills layer. A wiring may carry a skill at a box, so a manifest node may be a call to a protocol-layer tool, while nothing in the protocol layer depends on the skills layer for its evidence. Both are forward in presentation order and cross-cutting in practice, and each Part states them.

No definition has to be tracked across the presentation order to read this paper. Sections 3 to 6 restate every imported object at the point where it is first used here, and Figure 1 is a map of the Parts rather than a prerequisite chain.

The presentation order instead follows what a composite gains, not what a proof needs. Memory gives a composite an observation that survives further activity. Skills give it substitution of named units with a gate on the substitution. Protocols give it a discipline on connections, including on the case where nothing can be produced. The harness gives it deployment, admission and a record. Each is a strict addition to the previous ones, in the sense that the earlier layers have no vocabulary for it. That is the sense in which the composition is modular: a layer can be described, and its laws checked, without the layers above it.

7.6 Composition limits

The assembly is formal. The harness declares no dependency on two of the three component systems (Section 7.1), so the study does not establish that the implementations interoperate. Its preservation claims have the receipt-shaped scope stated in Section 2.2; they are not claims about running-system behaviour or a functor on all of **Arch**.

8 Agentic engineering

8.1 Engineering question

Cao gives a complexity argument for why externalization happens at all. Banu gives a categorical account of what externalization becomes once it has happened. The question this series set itself is what treating an agent as a lax monoidal functor $A: \mathbf{Arch} \rightarrow \mathbf{Sys}$ adds to the agent to result picture that a purely economic or capability account does not. And it asks at which of the counterexamples the added value stops. The question has to be asked of the fragments on which such a functor was actually constructed. Section 2.2 records that no interpretation of the whole of **Arch** exists in this series, so “an agent is a lax monoidal functor” is not among the things established here.

8.2 Categorical contribution

Preservation under composition. Cao’s argument is a counting argument. It says the number of interaction paths among n components grows exponentially while human capacity to reason about them does not, and concludes that decision logic must migrate somewhere that scales with compute. A counting argument cannot say which properties of two components survive putting them together,

because it has no notion of a property being carried along a map. The categorical account does, and Part IV’s results are of exactly that kind. A receipt-shaped certificate whose check factors through (G, Know) survives every translation; one that reads Φ need not (Part IV, Theorem 3.32). A translation that preserves every certificate can still change the admission behaviour, because the conjunction is antitone (Part IV, Proposition 3.34). Neither statement is available in a vocabulary whose only quantity is a count.

Decidable composition criterion. The sharper contribution is that the question “is composition structure-preserving here” has an answer that a practitioner can obtain without doing any category theory. Part IV’s criterion, strengthened here as Theorem 7.7, says that composition is structure-preserving exactly when the two components’ resource footprints are disjoint. For the system studied, that reduces to reading a migration file and seeing whether a fence column is drawn from one database sequence or from a per-row counter. The reduction runs both ways and is the useful part: a designer does not have to reason about monoidal functors to decide the question, and someone who knows what the answer settles does not have to read a schema. This is the one place in the series where the formalism produces a check that is cheaper than the thing it decides.

Capability-account limitation. If composing two agent applications creates ordering information that belongs to neither, then the composite is not understood by understanding the parts. That is the same difficulty Cao counts, restated as a property of a composition operator rather than as a number. The restatement is what makes it actionable: a number cannot be repaired, whereas a composition operator can be given a different sequencing resource. Part IV says exactly what that costs: a single cross-application allocation order. The identifier does not establish commit or replay order unless consumers sort by it and tolerate gaps and commit-order inversions.

8.3 Durable artifacts

Cao’s second claim is that the delivery chain collapses from artificial intelligence through software to result into agent to result, eliminating the software artifact as a necessary intermediary. On the same claim, the code an agent writes “is not the system; it is a transient artifact”. The evidence assembled by this series supports a narrower version of that claim and contradicts the broad reading, and the triple is what makes the difference statable.

What is transient in the three systems studied is the content a model produces inside a stage. What is durable is everything the triple names, and all of it is in version control:

- G is a set of declared manifests. In the harness, a manifest declares nodes, edges, roles, tools and policies, is loaded from disk by a named loader, and is content-addressed by a stable hash. In the protocol system, the corresponding artifact is a versioned wire contract with a closed operation vocabulary and an explicit version discriminator, mirrored by a generated contract for a second transport.
- Know is a set of schemas, policies, admission rules and retained receipts. In the harness these are hook subscriptions with a declared failure policy, deterministic idempotency keys, and receipts with a closed record shape. In the protocol system they are policy documents evaluated before planning and per-field provenance attached to answers. In the memory system they are a schema registry and structured-data validation, which Part I places on the Know side of the boundary precisely so that they are not confused with state.

- Φ is a pair of name-keyed resolution functions and a set of application manifests declaring adapters. It is data, it is deployed, and Banu’s Definition 2 makes it the thing a deployment is allowed to change.

So agentic engineering is not the end of durable artifacts. It is a change in which artifacts are durable: from the decision rules D of Cao’s Definition 2.1, which the model now writes at run time, to the wiring, the checks and the deployment, which people write and keep. This is a scoped correction rather than a refutation. Cao’s own three-generation table already assigns the complexity to the agent rather than to the artifact, and nothing here disputes that the emitted code is transient. What the triple adds is that the transient part is properly contained: it lives inside a stage, and the stage sits in a wiring that is declared, is checked, and is deployed by a map that is itself a parameter.

There is a corollary for practice that follows directly from Table 3. If the durable artifacts are G , Know and Φ , then the durable failure modes are failures of those artifacts. Every one this series found is a missing check on a declared object: an input port with no supplier, two siblings claiming one output key, a plan node that cannot state the hypothesis that would make its confinement result bear on authorization, a check that reads an environment variable instead of a declared port. None of these is a failure of a model, and none is repaired by a better model.

8.4 Role of the triple

Section 9 raises two questions: why retain the Architecture triple at all, and whether the categorical apparatus contributes anything to the composition analysis of Section 7. The two have different answers.

The apparatus is not doing the work in Section 7. Theorem 7.7 needs a partial composition law, a resource assignment and a fold, and none of the three components of the triple appears in its statement or its proof. A reader who wanted only that theorem could have it without reading a word about G , Know or Φ . Theorem 7.10 specialises it to the harness, but the specialisation uses only the stage set and the certificate set as index sets. That is less than the triple provides. So on questions about composition the triple is more structure than is needed, and the honest description of Section 7 is that it is a statement about partial monoids with a resource map which the four layers happen to instantiate.

The triple is doing the work elsewhere, and the place is identifiable. Part IV’s separation of preservation, that a check factoring through (G, Know) survives every translation while a check reading Φ need not, is a statement about *which component a predicate reads*. It has no formulation that does not name components, because its content is precisely that one component is left unconstrained by morphisms and the others are not, and that asymmetry is Banu’s Definition 2 rather than a modelling convenience. Remove the decomposition and the empirical finding it organises, that one check in the harness reads the deployment and the rest do not, has nothing to be a finding about. The same is true of the admission result, which needs Know to be a set a translation may enlarge. It is true too of the admissibility condition for the monoidal product, which is a condition on names in G and is therefore a statement about a specific code path rather than about an abstract monoid.

So the answer to whether the formalism is aspirational or wrong is neither, and the finding is sharper than either. The triple is load-bearing for questions of the form “on which component does this property depend”, and it is decorative for questions of the form “does this compose”. The division identifies which apparatus each class of question requires, and a reader who wants only one of the two kinds of question answered now knows which to carry.

8.5 Scope

On the two harness fragments of Section 2.2, the categorical reading makes preservation under composition a well-posed question and gives one criterion that is cheap to evaluate. The triple also identifies the durable wiring, checks and deployment data surrounding transient model-written code. These results do not extend to agents in general, predict model behaviour or preserve properties that no certificate states. Section 9 gives the counterexamples.

9 Limits and counterexamples

The interpretation fails at the four cases below; each claim names the Part that establishes it.

9.1 Single-axis memory

The correspondence’s memory row is accompanied by the claim that every fact carries both a valid time and a record time, which is what makes belief-state reconstruction possible: what did the agent know at decision point t . `ContextFS` does not support that. Its record type declares exactly two time-valued fields, both defaulting to a clock reading at call time. Its versioned model carries one timestamp per entry, not a pair. Its lineage operations write single-timestamp fields into a metadata map rather than a second axis.

The finding is stronger than a field count, and that is why it is a counterexample rather than a gap. As stated in Part I (Theorem 5.27), two histories differing only in valid time have the same recording, and their snapshot functions differ. So no function of the entire store recovers the distinction between a retroactive correction and a belated recording, and every bitemporal observation is degenerate in valid time. The system does answer “what did the system have on file at t ”, through a timeline lookup that compares against the single timestamp field (Part I, Corollary 5.28), and that is transaction time. The weaker distinction that the change-reason enumeration could have carried is not exposed on any agent-facing surface either (Part I, Remark 5.29).

Part I also shows the repair is small and that the categorical reading survives it. The single-axis coalgebra is a quotient of a bitemporal one, obtained by extending the record type and four operation signatures with one time argument, and the extended observation is not degenerate (Part I, Proposition 7.1). The gap is a design gap, not an obstruction. What the counterexample costs the correspondence is the headline feature of its first row. The most-quoted property of agent memory in that account is absent from the production system. It was absent for a reason the account does not surface: the operations timestamp themselves rather than being told when their content became true.

9.2 Open colour set

The skills row says skills are the operations of an operad whose colours are port types. The harness’s port carrier keys two maps by arbitrary strings, one carrying JSON values and one carrying artifact references, so ports are dynamically typed and unboundedly many rather than drawn from a small closed colour set. Part II states the idealisation required (Assumption 3.10) and makes it precise (Definition 3.2). The colour lattice exists, it has a top element, and almost every port carries the top, so a type system exists and is trivial almost everywhere (Part II, Remark 3.3).

The instructive part is that granting the idealisation entirely does not rescue the row. As stated in Part II (Example 5.11), renaming a consumer’s declared input port to a misspelling leaves every validation clause satisfied, so the manifest is accepted with a dangling input port. Under the colour

assumption both names carry the same colour, so the colours match. What fails is the existence of a supplier, which the operad reading requires and no code checks. Hence validation is not a sound recogniser of composites (Part II, Corollary 5.12). The openness of the colour set is not the obstacle. The absence of a linking check is. Part II locates a linking check of exactly the right shape one level up in the same repository (Code observation 6.7), which makes the counterexample a specification of a repair rather than only an objection.

Two further limits are not repairable by validation. A **Set**-valued algebra can represent deterministic correlations among several outputs but cannot represent nondeterministic results at a fixed input (Part II, Theorem 7.2). A PROP preserves the invocation as one multi-output operation but remains deterministic, and its interchange law still requires disjoint actual output domains and strand-local reads (Part II, Proposition 7.4). Adaptive coordination proposes operations after validation has run (Part II, Code observation 9.1), which is a boundary of the reading rather than a defect of it.

9.3 Coupled wiring and knowledge

The correspondence assigns protocols to G and the checkable material to Know. Neither system in this series draws that line cleanly, and the protocol system draws it in a way that is informative rather than merely untidy.

Its policy evaluator runs as a pre-planning gate, before the wiring exists, so the decision record is a precondition on whether a wiring may be built at all rather than something attached to a wiring. Its provenance layer attaches evidence to individual field values inside an executed plan's output, so that record is attached to the values travelling on the wires rather than to the wiring. Neither is a separately addressable component. The crate named for schemas, which a reading of the directory listing would have assigned this pillar's weight to, is an eighteen-line placeholder at the pinned commit (Part III, Section 7.2).

Part III is more precise than “the separation fails”. The system puts one fragment of the checkable material into the port types themselves, since the authorized field set is a leaf's port label. It carries the rest as values in the reason sets, with no third place (Part III, Remark 7.1). The consequence for the correspondence is stated there and adopted here. If the knowledge layer is distributed over the wiring's own type and reason vocabularies, then the triple is not recovered by reading off components. It is recovered, if at all, by a modelling choice about which part of G to call Know. The harness separates the two for its identified hook certificates, while it does not classify the per-operation `DagToolPolicy` envelope as a certificate (Part IV, Table 2, law L1). Neither system separates policy from wiring completely, so the degree of separation is a property of the system rather than of the correspondence, which makes the correspondence's usefulness system-dependent rather than universal.

9.4 Reference vocabulary

Banu's four-pillar table names concrete components of his own reference implementation: a bitemporal memory type and a run context for Memory, a skill stage and a pattern template for Skills, a wiring diagram for Protocols, and a skill organism for Harness. A recursive search for those six identifiers across the three repositories at the pinned commits returns no match, and each Part records the search for its own system (Part I, Remark 7.3; Part IV, Section 8.5).

The consequence is a restriction on what the series may claim, and it is a restriction the series accepts. None of the three systems implements that vocabulary. The claim is that systems built without knowledge of it independently exhibit a structure the vocabulary names, which is a weaker

claim than implementation and a more interesting one, and a reader should hold it to the weaker standard. In particular, nothing in this series is evidence that the reference implementation is correct, and nothing here transfers a measurement made on it.

There is a related structural point that belongs with this counterexample. The correspondence speaks of one wiring object; the series has two, and Part IV records a third difference on top of the erasure result: harness wirings are not tree wirings (Remark 3.2), because a manifest edge is a pair of endpoint lists and a manifest may declare several terminal stages, so the harness realises a strictly larger fragment of the wiring operad than the protocol layer does, and the protocol layer’s confinement argument, an induction over a tree, does not transfer. A single wiring object read four ways would not have this property. Its failure is the sharpest evidence that the four pillars are not four views of one thing.

9.5 Combined limitations

Banu states five limitations of his own: the framework is static, with no account of a harness evolving over time; there is a single reference implementation; certificate scope is limited to structural invariants rather than behavioural properties; the escalation experiment uses two models and one task; and the SWE-bench-lite result confirms a format-discipline ceiling at 8B parameters cross-model, not a task-resolution gain [1].

This series addresses exactly one of the five and worsens one other. It addresses the second by adding three independently developed systems, none of which was built from the framework, and by reporting for each of them which laws hold and which fail. It worsens the third: Banu’s certificates carry theorem statements because he wrote them that way, whereas every certificate found in the harness studied here is receipt-shaped and provably blind to everything except identity and record shape (Part IV, Proposition 3.7). So the certificate scope in the wild is narrower than the certificate scope in the reference implementation, and the two preservation results are not comparable quantities.

The other three limitations are untouched. This series contains no experiment, no measurement and no benchmark.

The repository-access limitation is stated in Section 1.2. Exact paths, identifiers, line ranges and quoted source excerpts make the code claims falsifiable for readers who have access; they do not provide an open artifact. The definitions and proofs are independent of that access.

9.6 Bibliographic corrections

Two bibliographic facts bear directly on the correspondence.

The paper on scaling coding agents via atomic skills, arXiv:2604.05013, has been withdrawn, the stated reason being that significant errors were discovered in the data after submission and affect the validity of the results [15]. That paper is one of the two empirical supports offered for the operadic reading of the Skills pillar: its finding that atomic skills compose without negative interference is read as evidence that operad composition supports property-preserving composition when the relevant certificate is closed under the operation. A withdrawn paper cannot carry evidential weight. No argument in this series depends on it, and Part II cites it only to identify what the correspondence appeals to. The consequence for the Skills row is that its empirical support is reduced to the reference implementation’s own measurements, and its structural support is what Part II establishes, which is three conditions holding, one holding conditionally, and three failing.

Two other attributions in the correspondence’s reference list do not match the arXiv records. The natural-language agent harness study, arXiv:2603.25723, is authored by Pan, Zou, Guo, Ni and

Zheng [19]; it is attributed there to “Erik Willstrom et al.”. That paper is the one read as validating the claim that harnesses are portable objects with algebraic structure, so the attribution matters for anyone following the citation. The skill benchmarking paper, arXiv:2603.28815, is authored by Wang, Wang and Xu [28]; it is attributed there to “Zhiyu Chen et al.”, and no author of that name appears on the record. This series uses the arXiv-listed authors in both cases. Neither correction affects any argument; both affect whether a reader can find the work.

9.7 Interpretive limits

Collecting the four counterexamples, the interpretation stops paying at three distinguishable kinds of place.

It stops where the formalism names a gap that a schema inspection finds faster. The bitemporal gap is the clearest case. A coalgebraic reading of memory does not force a second time axis, and counting the time-valued fields on the record type would have found the same thing in a minute. What the formalism added was the separation statement, which says the distinction is not merely absent but unrecoverable from anything in the store, and the quotient statement, which says the lineage machinery survives the repair unchanged. Those are worth having and they are not what the correspondence promised.

It stops where the mapping from pillar to component is a modelling choice rather than a reading. Where a system distributes its checkable material over its port types and its reason sets, there is no component to point at, and the assignment of a pillar to a component becomes a decision made by whoever is doing the assigning. An account whose central move is such an assignment cannot then use the assignment as evidence.

It stops where the chosen semantic category is too poor for the system. A **Set**-valued algebra represents deterministic correlations but not nondeterministic output tuples at a fixed input. Passing from an operad to a PROP preserves multi-output invocation syntax without removing that semantic limit, and introduces an interchange law that can fail when output domains overlap. This is a limit of the proposed semantics, not of the runtime.

The formalism nevertheless yields the following results. The stable-ancestry result and the characterisation of the fragment it holds on; the layer-wise disjointness condition and its recurrence in the interchange law; the trace-monoid soundness of the effect order; the confinement theorem and the identification of the two node cases that carry it; the algebra of refusals over strict tree wirings and the counterexample showing why the restriction is needed; the separation of Φ -free from Φ -reading checks; and the strength criterion in the resource form of Theorem 7.7. Each is a statement about a production system that a maintainer can act on.

9.8 Non-vacuity

The four counterexamples invite a stronger objection: that the correspondence has been scoped until it says nothing. The objection has a testable form. An account is empty in the relevant sense if it identifies nothing a competent reading of the source would not have identified, and the Parts separate their findings on exactly that line. Part II says plainly that its first failed condition needed no formalism, and then lists four findings that did: a linear-time condition on manifests that no reading of the runtime as dynamically typed would suggest looking for, the same condition recurring in the interchange law of the richer structure, a loss no validation clause could repair, and a level at which the runtime is sound, exhibited by naming the monoid rather than by asserting soundness. Part III’s confinement theorem shows which two of eight plan cases carry the induction, which is not visible from the function that computes the field set. Part IV’s separation predicts, before any check

is examined, which class of check will fail to survive redeployment, and one check in the system falls in the predicted class. The prediction that the account is empty therefore fails on at least six statements, each attached to a production system and each actionable. What the counterexamples do establish should not be softened: the correspondence is not a reading-off of components, its first row is missing its advertised feature, and its notion of preservation degenerates to identity for the certificates that exist in the wild.

10 Comparison with the compiler functors

Banu’s validation uses five compiler functors and is the part of his paper closest to the questions Part IV asks [1]. The comparison below introduces no new experiment or measurement.

10.1 Compiler checks

A compiler is presented as a functor $F: \mathbf{Arch}(\text{Source}) \rightarrow \mathbf{Arch}(\text{Target})$ producing a preservation result with three checks: graph preservation, asking that the source stage names and edges appear in the target; a certificate replay invariant, asking that source certificate theorems, parameters and evidence are preserved and that the verifier accepts; and deployment-map preservation, asking that source stage names are present in the target, with the mode mapping allowed to differ. Five targets are reported: Swarms, DeerFlow, Ralph, Scion and LangGraph. The LangGraph compiler is described as per-stage, emitting one target node per source stage, each calling the same single-stage execution method the native runtime uses, and parallel stage groups compiling to a fork and join through the target’s fan-out interface; this design is introduced explicitly to avoid what is called the reimplementing trap, an earlier attempt having needed ten rounds of correction for behavioural parity. The reported outcomes are that across the five targets, three supported certificate types preserve identity and verify in the reported runs, and that across the four non-LangGraph compilers, three of three certificates were preserved and three of three verified when compiling organisms carrying several certificates [1].

10.2 Modeled checks

The three checks correspond closely to the morphism conditions Part IV writes down. Graph preservation is the sub-wiring embedding; certificate replay is the pair of claim identity and check acceptance; and the deliberate weakness of the deployment check, asking only that names appear and allowing the mode mapping to differ, is exactly the decision not to constrain Φ . Part IV takes that decision from Banu’s Definition 2 [1] and then shows what it costs (Proposition 3.11): on the subcategory of deployment-independent checkers, forgetting the deployment map is an equivalence. That result gives no guarantee for a check that reads deployment data a compiler may change.

Part IV’s Theorem 3.32 proves replay stability for a receipt-shaped check that factors through (G, Know) and gives a counterexample for a check that reads Φ . Banu describes his measured certificates as priority gating, absence of false activation and absence of oscillation [1]. Reading those descriptions as checks on wiring is an inference, because this study did not inspect their checker implementations. Under that inference, their reported preservation is consistent with the theorem. It provides no evidence about deployment-reading checks.

The compilers’ preservation result is certificate-wise, and certificate-wise preservation is silent about the conjunction. As stated in Part IV (Proposition 3.34), the admission predicate is antitone in the certificate set, so a compiler that preserves every source certificate can still turn an admitted operation into a refused one, simply by having more certificates in the target. This is not a defect

of the imported preservation proposition, which is a statement about certificates; it is a limit on what an operational use of that proposition can conclude, and a preservation result that checks each certificate separately does not detect it. In the harness studied here, installing one additional admission-phase hook is precisely such a translation.

10.3 Comparison limits

Two differences make the two bodies of evidence incomparable rather than in tension.

The certificates are not the same kind of object. Banu’s carry theorem statements because his reference implementation writes them that way; the harness studied here carries a schema string, identity fields and a retained body, and Part IV’s Proposition 3.7 shows such a check is invariant under any relabelling of payloads fixing the identity fields. Preserving a theorem-carrying certificate and preserving a receipt are different achievements, and no rate computed over one transfers to the other.

The compositions are not the same operation. A compiler translates one architecture into another and is a morphism of **Arch**. The composition Part IV studies is the monoidal product, two architectures installed side by side, which places no wires between the summands. The laxness result is about the second and says nothing about the first. A reader who takes “certificate preservation holds at 100 percent” as evidence that composition is well behaved is conflating translation with composition; the two questions are independent, and this series answers only the second.

One prior case study belongs beside the comparison. An earlier set of papers by the present author describes a different harness in a different runtime, an umbrella application composing a scheduler, a tool interface, a memory layer and a planner engine through a dependency-ordered supervision tree, and argues that every pairwise combination of subsystems yields capabilities neither has alone [9]. That is the composition-creates-structure claim made informally, and Theorem 7.7 is one precise instance of it. The numbering of that series is unrelated to this one, and its Part IV is a planner engine rather than a harness.

11 Open problems

11.1 Problems for the account

1. *One interpretation, or two.* The series constructs a lax interpretation on the durability fragment and a strong receipt-sequencing abstraction on the hook fragment, and no interpretation on their union. Settled by: a lax monoidal functor on the full subcategory generated by both, together with a determination of whether it is strong, lax, or neither on architectures whose certificate sets mix the two kinds; or a proof that no such functor extends both.
2. *Is the disjointness pattern a theorem.* Theorem 7.1 says the four side conditions have one shape and Theorem 7.2 says the shape carries a partial commutative monoid. It does not say that the four instances are instances of a single statement with content, and Theorem 7.3 says why they may not be. Settled by: either a theorem whose four instances are the four rows of Table 3, with a nontrivial conclusion in each, or an argument that the name sets are too different for one to exist.
3. *Reflection at the lower layers.* Theorem 7.7 applies where a target object carries a relation presented as the kernel of a resource assignment. Only the harness layer presents its target that way, and Theorem 7.12 records that the memory layer’s correct support is a resource footprint arrived at by inspection. Settled by: a resource-indexed interpretation of the memory layer

whose comparison map fails to be invertible exactly in the situation of Part I’s Proposition 6.4, or an argument that the change of an observation value is not a relational phenomenon and so lies outside the theorem’s reach.

4. *A theorem field for a harness certificate.* Receipt-shaped certificates make preservation an identity statement. Settled by: a claim language in which a check can state what its receipt attests, together with a checker that evaluates the claim against the receipt rather than against the request’s identity fields, and a demonstration that the resulting certificates are still decidable and still transported by the morphisms of **Arch**.
5. *Typed wiring across the layer boundary.* The type erasure from the protocol layer’s port types to the harness’s manifest keys has no manifest-level section, because a manifest edge records node identifiers and no port datum. Settled by: an edge type carrying port names on both ends, together with a validation clause relating a declared input to a producer’s declared output, and a determination of whether the resulting map is a section of the erasure.

11.2 Required system changes

ContextFS. Add a valid-time field to the record type and an optional valid-time argument to the four derivation operations and to the three agent-facing surfaces, and compare against it in the timeline lookup; Part I shows the lineage machinery is untouched by this and that the resulting observation is not degenerate. Decide what merge does with valid time when its arguments disagree, which the model does not force. Make identifier resolution total or scope it, since resolution by prefix is what breaks the frame property and the documented interface invites abbreviation. Reconcile the two ancestry representations, and state what a reconciliation does when they conflict. Give deletion a tombstone semantics under which the ancestry-immutability result has an analogue.

AgentHero. Declare an initial input interface and require every node input to come from that interface or from exactly one ancestor output. Confine each handler’s returned value keys to its declared outputs, then reject sibling collisions; declared disjointness alone does not control undeclared returned keys. Decide whether the durability fence should come from one sequence or from many, and state what the global allocation order buys against the strength it costs. Move the deployment-relative part of the artifact containment check into the wiring, by making the authorized root a declared port type rather than an ambient environment variable, which is the remedy the protocol system already implements for authorized field sets. Give the receipt a claim it attests, per the theorem-field problem of Section 11.1.

CatDB. Make the confinement hypothesis dischargeable where the confinement result is proved: at present the crate that establishes field confinement does not depend on the policy crate, so it cannot state the leaf constraint under which the result bears on authorization. Give port labels base types, so that a join bringing together two identically named fields is a type-level question rather than a value-level one resolved by provenance. Move the schema semantics into the crate named for them, or rename the crate. Decide whether to allow shared subplans, which would require the confinement induction restated over a directed acyclic graph with memoisation in the field computation, and would move the realised fragment from the tree sub-operad to the full wiring operad.

11.3 Falsification criteria

A system in which the four pillars are present and the triple cannot be recovered even by a modelling choice would falsify the interpretation directly. Section 9.3 is the closest this series came: there the triple is recoverable only by a choice, which is a weakening rather than a refutation.

A resource-indexed interpretation satisfying Theorem 7.6, including a bijective comparison and the fixed-resource equations, would refute Theorem 7.7 if its factors had disjoint resource footprints but its comparison were noninvertible, or if overlapping footprints produced an invertible comparison.

The pinned source observation remains falsifiable by finding a behavioural claim field in one of the examined certificate types at commit `1c3ad24`. A later system with behavioural certificates would not refute that observation; it would instead defeat any broader extrapolation that harness certificate preservation is necessarily an identity statement.

12 Conclusion

The four pillars admit a structure-preserving interpretation only on named fragments with explicit support conditions. The component papers identify those conditions and their failures: ancestry is stable only on the monotone memory fragment; handler operations require an identity beyond their port signatures; disjoint actual output domains suffice for order-independent skill execution; protocol encoding failures must remain typed reasons; and deployment-relative checks require their own preservation argument. The production systems satisfy some of these obligations and violate others.

The resource theorem gives the reusable result. For an interpretation whose comparison differs from the factors only by cross-factor resource relations, the comparison is invertible exactly when the two factor footprints are disjoint. This can be checked before composition. It does not predict model behaviour or make unchecked properties compositional. The categorical account earns its cost when it exposes the names, resources, and certificates on which a preservation claim depends; outside that role, ordinary dependency graphs and source-level evidence are enough.

A Evidence map

This paper makes no code claim of its own. Every code-backed claim in the main text belongs to a Part, is established there by reading a named file at a pinned commit, and appears as a row of that Part’s code evidence appendix. Table 4 maps each such claim to the Part, to the numbered result and source label that state it, and to the repository and commit whose source carries the evidence. Column two names the Part by its position in the series, and the four are [10, 11, 12, 13], carrying the identifiers `GrokRxiv:2026.09.memory-coalgebra`, `GrokRxiv:2026.09.skills-operad`, `GrokRxiv:2026.09.protocols-wiring` and `GrokRxiv:2026.09.harness-architecture` respectively. The file, identifier and line range for every row are in the code evidence appendix of the Part named, under the label given in column three. One row is an absence claim, which no line of source exhibits; Part IV records it in the preamble of its own appendix as the exact recursive search that establishes it, and that row is marked accordingly.

Table 4: Evidence map: every code-backed claim in this paper, with the Part, the numbered result and source label, the repository and the commit that carry its evidence.

Claim as used in this paper	Part	Result and label in that Part	Repository	Commit
The record type carries two clock-valued fields and no valid time, so valid time is unrecoverable from the store (Section 9.1)	I	Thm. 5.27, thm:lineage-not-bi-temporal	ContextFS	a93035d
The timeline lookup compares against a single timestamp field, so it answers a transaction-time question (Section 9.1)	I	Cor. 5.28, cor:timeline-transaction	ContextFS	a93035d
The change reason is not exposed on any agent-facing surface (Section 9.1)	I	Rem. 5.29, rem:reason-not-exposed	ContextFS	a93035d
Derivation calls attach ancestry-labelled edges only to constructed identifiers, which is what makes ancestry stable; freshness of those identifiers is carried as a hypothesis and is sampled rather than checked (Section 3)	I	Lem. 5.6, lem:contextfs-schema ; Thm. 5.7, thm:ancestry-immutable ; Prop. 5.3, prop:freshness-unchecked	ContextFS	a93035d
Save under an existing identifier replaces, and deletion removes edges in both directions, so the stability result is restricted to the derivation fragment (Section 3)	I	Prop. 5.2, prop:save-delete-nonmonotone	ContextFS	a93035d
The merged type is decided by an enumeration order the interface does not declare (Section 3)	I	Prop. 5.20, prop:merge-underdetermined	ContextFS	a93035d
Merge takes the namespace of its first argument and unions metadata across all of them, carrying payload across a namespace boundary (Section 3)	I	Prop. 5.18, prop:namespace-not-hom	ContextFS	a93035d
Ancestry is written twice, into record metadata and into the edge set, and the two writes are not guarded alike (Section 3)	I	Prop. 5.14, prop:two-representations	ContextFS	a93035d
Edge endpoints are unchecked by default, so a store with a dangling ancestry edge is reachable (Section 7.3)	I	Prop. 5.30, prop:no-referential-integrity , Prop. 6.4, prop:union-heals	ContextFS	a93035d
Identifier resolution is by prefix without ordering, so the frame property fails (Section 3)	I	Prop. 6.3, prop:frame-failure , Rem. 6.8, rem:birthday	ContextFS	a93035d
Identifier resolution filters on the identifier and the deletion field only, so namespaces do not supply support disjointness (Table 3)	I	Rem. 6.7, rem:isolation ; Prop. 6.3, prop:frame-failure	ContextFS	a93035d
The schema registry and structured-data validation are a checking layer, not part of the state (Section 8.3)	I	Sec. 6.4, sec:with-harness	ContextFS	a93035d

Claim as used in this paper	Part	Result and label in that Part	Repository	Commit
The nearest run-scoped object is a session, and the reference vocabulary's run context does not occur (Section 9.4)	I	Rem. 7.3, rem:not-runcontext	ContextFS	a93035d
The port carrier keys two maps by arbitrary strings, one to JSON values and one to artifact references (Section 9.2)	II	Def. 3.2, def:colour-set ; Obs. 5.10, obs:partial-colour	AgentHero	1c3ad24
Validation never relates a declared input to a producer, so an accepted manifest may have a dangling input port (Section 9.2)	II	Ex. 5.11, ex:open-colour-set , Cor. 5.12, cor:not-recogniser	AgentHero	1c3ad24
The store merge overwrites on a repeated key, so sibling order matters exactly when output keys collide (Table 3)	II	Obs. 5.16, obs:order , Ex. 5.17, ex:merge-collision	AgentHero	1c3ad24
The execution layering depends on the dependency relation alone (Section 4)	II	Obs. 5.3, obs:layering , Prop. 5.4, prop:L3	AgentHero	1c3ad24
Concurrently scheduled nodes are checked for interference by a derived concurrency class (Section 4)	II	Obs. 5.6, obs:L6 , Def. 3.13, def:effect-grade	AgentHero	1c3ad24
A node with empty declared inputs may read the whole store (Section 4)	II	Obs. 5.14, obs:arity	AgentHero	1c3ad24
The kind vocabulary contains no identity node (Section 4)	II	Obs. 5.20, obs:no-unit	AgentHero	1c3ad24
The loop construct is a bounded unrolling rather than a trace (Section 4)	II	Obs. 6.4, obs:trace	AgentHero	1c3ad24
The plugin capability resolver performs the linking check the graph layer omits (Section 9.2)	II	Obs. 6.7, obs:capability-linking	AgentHero	1c3ad24
Coordination is declared with bounded team parameters and worker role templates that manifest edges do not connect (Section 9.2)	II	Obs. 9.1, obs:coordination	AgentHero	1c3ad24
Declared colours are not enforced colours in the secondary system (Section 9.2)	II	Ex. 9.2, ex:declared-not-enforced	agent-os	61cb399
The typed-outcome and refusal vocabulary distinguishes a successful empty answer from each way of declining (Section 5)	III	Def. 3.10, def:outcome-type , Tab. 2, tab:audit	CatDB	7cc9341a
Plan validation is hereditary and the field computation confines root fields to leaf fields (Section 5)	III	Prop. 5.1, prop:hereditary , Thm. 5.2, thm:field-confinement	CatDB	7cc9341a
The crate proving confinement does not depend on the policy crate, so it cannot state the leaf hypothesis (Section 11.2)	III	Rem. 5.4, rem:hypothesis-elsewhere	CatDB	7cc9341a
Policy is a pre-planning gate and provenance is attached to field values, so neither is a separately addressable component (Section 9.3)	III	Rem. 7.1, rem:know-in-ports	CatDB	7cc9341a

Claim as used in this paper	Part	Result and label in that Part	Repository	Commit
The crate named for schemas is an eighteen-line placeholder (Section 9.3)	III	Sec. 7.2, sec:limit-stub	CatDB	7cc9341a
Port labels are field-name sets carrying no base type (Table 1)	III	Sec. 7.3, sec:limit-names	CatDB	7cc9341a
Plan variants own their inputs, so the realised fragment is the tree sub-operad (Section 5)	III	Sec. 7.4, sec:limit-tree	CatDB	7cc9341a
A union whose inputs pin the same source is refused (Table 3)	III	Prop. 5.5, prop:source-linearity	CatDB	7cc9341a
A manifest edge records node identifiers and no port datum, so the type erasure has no manifest-level section (Table 2)	III	Prop. 6.1, prop:type-erasure , Cor. 6.2, cor:composite-loses-w1	AgentHero	1c3ad24
Certificates in the harness carry a schema string, identity fields and a retained body, and no theorem (Section 6)	IV	Def. 3.6, def:receipt-shaped , Sec. 8.1, sec:limit-receipt	AgentHero	1c3ad24
The receipt check is total, decidable and identity-based (Section 6)	IV	law L2 in Tab. 2, tab:laws	AgentHero	1c3ad24
Hook subscriptions are separately addressable and the per-operation policy envelope is declared inside the manifest (Section 9.3)	IV	law L1 in Tab. 2, tab:laws	AgentHero	1c3ad24
Admission refuses on executor error, invalid receipt and unpinned bundle (Section 6)	IV	law L4 in Tab. 2, tab:laws	AgentHero	1c3ad24
One check reads the workspace root and an environment variable, so it reads the deployment map (Section 6)	IV	law L6 in Tab. 2, tab:laws , Sec. 8.3, sec:limit-phi	AgentHero	1c3ad24
The registry refuses two applications claiming the same type name, which is the admissibility condition for the monoidal product (Table 3)	IV	Prop. 3.21, prop:tensor-admissibility , law L7 in Tab. 2, tab:laws	AgentHero	1c3ad24
The durability fence is derived from a database sequence with no application scoping, which makes the composite’s sequencing relation total (Section 2.2)	IV	Thm. 6.3, thm:laxness	AgentHero	1c3ad24
Hook delivery fences are per row, which makes that fragment strong (Section 2.2)	IV	Prop. 6.7, prop:hook-strong	AgentHero	1c3ad24
A manifest edge is a pair of endpoint lists, so one stage may supply several consumers and harness wirings are not tree wirings (Section 9.4)	IV	Rem. 3.2, rem:not-tree	AgentHero	1c3ad24
Hook records give names and payloads distinct typed record positions, so the sorting hypothesis holds for them (Table 1)	IV	Rem. 3.4, rem:sorting	AgentHero	1c3ad24

Claim as used in this paper	Part	Result and label in that Part	Repository	Commit
Deployment is realised by name-keyed provider resolution and application adapter dispatch (Section 8.3)	IV	Sec. 5.5, <code>sec:phi-code</code>	AgentHero	1c3ad24
The workspace manifest and the harness runtime package declare no dependency on either of the other two systems (Section 7.1)	IV	Sec. 8.7, <code>sec:limit-integration</code>	AgentHero	1c3ad24
The six identifiers of the reference implementation’s four-pillar table return no match (Section 9.4)	IV	Sec. 8.5, <code>sec:limit-operon</code> ; recorded as a search, not a row	AgentHero	1c3ad24
A prior harness composition case study in a different runtime states the same composition-creates-structure claim informally (Section 10)	IV	Sec. 9, <code>sec:related</code>	agent-os	61cb399

References

- [1] B. Banu. Harness engineering as categorical architecture: structural guarantees are harness-level properties. arXiv:2605.12239, 2026.
- [2] Z. Cao. Agentic software: how AI agents are restructuring the software paradigm. arXiv:2606.05608, 2026. The version consulted here is titled “The end of software engineering: how AI agents are fundamentally restructuring the software paradigm”.
- [3] G. Deng, Z. Chen, Z. Yu, et al. SWE-Milestone: evaluating AI agents on continuous software evolution. arXiv:2603.13428, 2026. Titled “EvoClaw” in the version cited by [2]; the title given here is the one on the current arXiv record.
- [4] B. Fong and D. I. Spivak. Seven sketches in compositionality: an invitation to applied category theory. arXiv:1803.05316, 2018.
- [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] R. Kumar and P. Ramagopal. Agentic engineering: how swarms of AI agents are redefining software engineering. LangChain Blog, April 2026. <https://www.langchain.com/blog/agentic-engineering-redefining-software-engineering>.
- [8] Q. Liu. λ_A : a typed lambda calculus for LLM agent composition. arXiv:2604.11767, 2026.
- [9] M. Long. The AI operating system, Part V: modular synthesis. `agent-os` repository, `papers/latex/synthesis.tex`, commit 61cb399, 2025. The numbering of that series is internal to `agent-os` and unrelated to the present one.
- [10] M. Long. Coalgebraic Memory. Part I of the Agentic Engineering series, The YonedaAI Collaboration, YonedaAI Research Collective, 2026. GrokRxiv:2026.09.memory-coalgebra.

- [11] M. Long. Operadic Skill Composition. Part II of the Agentic Engineering series, The YonedaAI Collaboration, YonedaAI Research Collective, 2026. GrokRxiv:2026.09.skills-operad.
- [12] M. Long. Typed Protocol Wiring. Part III of the Agentic Engineering series, The YonedaAI Collaboration, YonedaAI Research Collective, 2026. GrokRxiv:2026.09.protocols-wiring.
- [13] M. Long. Harness Architecture. Part IV of the Agentic Engineering series, The YonedaAI Collaboration, YonedaAI Research Collective, 2026. GrokRxiv:2026.09.harness-architecture.
- [14] Y. Ma, R. Cao, Y. Cao, et al. Lingma SWE-GPT: an open development-process-centric language model for automated software improvement. arXiv:2411.00622, 2024.
- [15] Y. Liu. Scaling coding agents via atomic skills. arXiv:2604.05013, 2026. Withdrawn; the submission history records that significant errors were discovered in the data after submission and affect the validity of the results. The author given here is the one on the current arXiv record; the reference list of [1] gives a longer one. Cited only to identify a claim made elsewhere; no argument in this paper depends on it.
- [16] S. Mac Lane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics 5. Springer, 2nd edition, 1998.
- [17] Q. Meng, Y. Wang, L. Chen, et al. Agent harness for large language model agents: a survey. Preprints, 2026. DOI 10.20944/preprints202604.0428.v2.
- [18] Model Context Protocol contributors. Model Context Protocol specification. <https://modelcontextprotocol.io/specification/2025-06-18>, 2025.
- [19] L. Pan, L. Zou, S. Guo, J. Ni, and H.-T. Zheng. Natural-language agent harnesses. arXiv:2603.25723, 2026. Attributed to “Erik Willstrom et al.” in the reference list of [1]; the authors given here are those on the arXiv record.
- [20] 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.
- [21] D. Rupel and D. I. Spivak. The operad of temporal wiring diagrams: formalizing a graphical language for discrete-time processes. arXiv:1307.6894, 2013.
- [22] 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.
- [23] R. T. Snodgrass. *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann, 2000.
- [24] D. I. Spivak. The operad of wiring diagrams: formalizing a graphical language for databases, recursion, and plug-and-play circuits. arXiv:1305.0297, 2013.
- [25] OpenAI Preparedness team. Introducing SWE-bench Verified. <https://openai.com/index/introducing-swe-bench-verified/>, 2024. Dataset at https://huggingface.co/datasets/SWE-bench/SWE-bench_Verified.
- [26] V. Trivedy. The anatomy of an agent harness. LangChain Blog, March 2026. <https://www.langchain.com/blog/the-anatomy-of-an-agent-harness>.

- [27] D. Vagner, D. I. Spivak, and E. Lerman. Algebras of open dynamical systems on the operad of wiring diagrams. *Theory and Applications of Categories*, 30:1793–1822, 2015. arXiv:1408.1598.
- [28] L. Wang, Z. Wang, and A. Xu. SkillTester: benchmarking utility and security of agent skills. arXiv:2603.28815, 2026. Attributed to “Zhiyu Chen et al.” in the reference list of [1]; the authors given here are those on the arXiv record.
- [29] 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.