

Harness Architecture

Part IV of *Agentic Engineering*

Matthew Long

The YonedaAI Collaboration, YonedaAI Research Collective

Chicago, IL

`matthew@yonedaai.com`

September 2026

Abstract

An agent harness surrounds a language model with tools, memory, wiring and policy. Banu proposes that this layer is already described by the ArchAgents Architecture triple: a wiring, a set of certificates and a deployment map, with agents as monoidal functors interpreting architectures in concrete systems. We test the proposal against one production control plane, **AgentHero** at commit `1c3ad24`. Each certificate carries a scope in the wiring and a decidable replay check. The target category records which receipts a store design draws from a common monotone sequence, and nothing that a run could change. The interpretation is lax on the durability fragment, where the comparison map is a bijection with no inverse because one database sequence orders receipts of independently installed applications. The receipt-sequencing abstraction is strong on the hook delivery fragment, whose fences are per row; it does not model scope-based event eligibility. One criterion covers both abstractions: strength holds exactly when the sequencing resource is local to a component. A certificate whose check reads only the wiring and the certificate set survives every architecture morphism; one that reads the deployment map need not. The source contains one of each. Seven laws for a harness realization are settled against the code: five hold, one partially, and one fails. The certificates are receipt-shaped rather than theorem-shaped, which bounds what preservation can mean here. At this commit the harness ships the skills runtime and declares no dependency on the memory or protocol systems, so the composition of the three pillars is a claim about shape, not about the artifact.

Contents

1	Introduction	3
1.1	Scope	3
1.2	Contributions	4
1.3	Series context	5
2	Background	5
2.1	The harness pillar	5
2.2	The Architecture triple	6
2.3	Monoidal structure	6
3	The formal model	7
3.1	Harness wirings	7

3.2	Certificates	8
3.3	Architectures and their morphisms	9
3.4	The monoidal product	11
3.5	Systems	12
3.6	Interpretations	14
3.7	Replay stability	14
4	Harness laws	16
5	The system	17
5.1	Model-to-source mapping	17
5.2	The knowledge layer	17
5.3	Admission	20
5.4	Fenced delivery	20
5.5	The deployment map	22
5.6	Registry composition	22
5.7	The harness runtime boundary	23
6	Law checking	24
6.1	Law 1: separability	24
6.2	Law 2: receipt identity replay	24
6.3	Law 3: deterministic replay identity	25
6.4	Law 4: fail-closed admission	25
6.5	Law 5: fence exclusion	25
6.6	Law 6: Φ -independence	26
6.7	Law 7: name disjointness	26
6.8	Laxness on the durability fragment	26
6.9	Strength of the hook-fence abstraction	29
7	Layer composition	30
7.1	Layer contribution	30
7.2	Part I dependency	30
7.3	Part II dependency	31
7.4	Part III dependency	31
7.5	Dependency graph	32
8	Limits and counterexamples	32
8.1	Receipt-shaped certificates	32
8.2	Sorting hypothesis	32
8.3	Redeployment counterexample	33
8.4	Lax composition	33
8.5	Reference vocabulary	33
8.6	Fence scope	33
8.7	Missing system composition	33
9	Related work	34
10	Conclusion	35

1 Introduction

An agent harness is what remains of an agent system when the language model is removed. It holds the prompts, the tool surface, the memory, the wiring between stages, the policy that admits or refuses an action, and the machinery that decides which model or process runs where. Recent surveys treat this layer as a subject in its own right and enumerate its components [9, 19]. An industry account states the division of labour in the same terms: the model holds the capability and the harness is the system that makes it useful [16]. Enumeration is not composition. A list of six components does not say what happens when two harnesses are installed side by side, which properties of a harness survive translation into a different runtime, or what it would mean to check that they do.

Banu [2] argues that a formal answer already exists. The ArchAgents programme of de los Riscos, Corbacho and Arbib [4] models an architecture as a triple (G, Know, Φ) . The syntactic wiring G is a graph of modules, ports and directed edges. The knowledge structure Know holds invariants and certificates. The deployment map Φ assigns concrete implementations to abstract capability slots. Morphisms are structure-preserving translations, which in practice are compilers. Banu maps Zhou et al.’s four externalization pillars onto this triple: Memory as coalgebraic state, Skills as operad-composed objects, Protocols as the wiring G , and Harness as the full triple. He validates the mapping with five compiler functors and a reference implementation of his own.

This Part takes the fourth row of that table seriously and asks whether a harness built independently of that programme instantiates it. The system is **AgentHero**, a control plane written in Rust with a JavaScript harness runtime, at commit `1c3ad24`. It was not built from Banu’s paper: none of the reference implementation’s identifiers occurs anywhere in it (Section 8). If the triple is a real description of harnesses rather than a description of one harness, the structure should be recoverable anyway.

1.1 Scope

What we claim is narrower than the correspondence taken at face value, and we state it in full before anything is proved. We do not claim that the four externalization pillars and the Architecture triple (G, Know, Φ) are the same thing. We claim three things, under assumptions stated at the point where each is used. There is a structure-preserving interpretation of the four pillars in (G, Know, Φ) . Each of the three systems examined in this series realises that interpretation on an identifiable fragment, which for the harness is the fragment delimited law by law in Section 6. The interpretation fails at identifiable points, of which this Part exhibits several in Section 8. Memory, Skills, Protocols and Harness is the order in which the series presents the pillars, not an order of logical dependency, and where a dependency runs against it we say so (Section 1.3).

For the Harness pillar this specializes to three questions. Does the harness runtime carry all three components of the triple, in separable form? On which fragment can an interpretation **Arch** $\rightarrow$ **Sys** be stated and checked to be lax monoidal? And what can “certificate preservation” mean for a harness whose certificates are receipts and fence tokens rather than triples of theorem, parameter map and replayable derivation?

The last of the three is where the counterexample lives, and its short answer governs the whole Part. Banu’s Definition 1 asks a certificate to carry a theorem statement τ , a symbol-to-parameter

map σ , and evidence *evds* that can be mechanically replayed. The harness studied here carries schema strings, identity keys and retained receipt bodies. There is no theorem. Preservation therefore cannot mean that a theorem survives translation. It can only mean that a decidable check keeps accepting the same transported witnesses. Our name for that is replay stability, defined precisely in Theorem 3.30, and it is never called verification here. Every statement about the running system below is an implementation-level claim established by reading the source at the pinned commit, of the form “the implementation at commit `1c3ad24` performs Y ”. No claim in this paper rests on an observed run, and Theorem 3.24 is built so that the interpretation into systems has no way to depend on one.

1.2 Contributions

1. A definition of the category **Arch** of Architecture triples (Theorems 3.3, 3.8 and 3.9), in which a certificate carries a scope in the wiring and a decidable replay check. The deployment map is a parameter rather than preserved data, following Banu’s Definition 2. **Arch** is shown to be a category (Theorem 3.10) and symmetric monoidal under disjoint union of wirings (Theorem 3.18).
2. A target category **Sys** (Theorem 3.24) whose objects carry a sequencing relation rather than fence values, so that a functor into it is defined by a schema and never by a trace (Theorem 3.23). An interpretation is then a lax monoidal functor (Theorem 3.29).
3. A precise statement of the cost of treating the deployment map as a free parameter. On deployment-independent checks, forgetting Φ is an equivalence of categories (Theorem 3.11), so no functorial guarantee about deployment can be derived in **Arch**. The deployment-preserving refinement **Arch** ^{Φ} (Theorem 3.13) is a genuinely different category, and the deployment-changing counterexample used by the separation theorem is not a morphism in it.
4. A separation theorem for certificate preservation (Theorem 3.32): a certificate whose check factors through (G, Know) is replay-stable under every architecture morphism, while a certificate whose check reads Φ need not be. Both kinds occur in **AgentHero**, and we name them.
5. A proof that certificate-wise preservation does not imply preservation of the derived fail-closed admission predicate (Theorem 3.34), which is antitone in Know . This is a gap in the operational reading of certificate preservation, not in the underlying proposition.
6. A laxness result (Theorem 6.3) locating the failure of strength in one concrete artifact. The harness draws every durability fence from a single database sequence, so the comparison map is a bijection that has no inverse in **Sys**. On the hook delivery fragment, whose fence tokens are per row, the receipt-sequencing abstraction is strong (Theorem 6.7). It does not model which certificate-event pairs are eligible, so it answers the fragment question only for fence allocation.
7. An admissibility result for the monoidal product (Theorem 3.21): the registry’s refusal to install two applications claiming the same stage name is exactly the side condition that makes $\otimes$ well-defined on globally named architectures.
8. Seven laws for harness realizations (Section 4) and their status against the source at the pinned commit (Section 6), with five satisfied, one partial and one violated, together with a code evidence table (Section A).

1.3 Series context

This Part is the last of four and has the heaviest import list. It cites the sibling Parts rather than restating them.

From Part I [10] it imports the memory coalgebra $(S, \alpha: S \rightarrow FS)$ and the narrowing that ContextFS’s lineage is single-axis rather than bitemporal. Part I’s result is used here only to keep Know and S apart: a schema registry or a validated manifest is a certificate, a stored memory record is not.

From Part II [11] it imports the coloured skills operad $\mathcal{O}$, the reading of a validated DAG manifest as a formal composite, and the statement that AgentHero’s ports are string-keyed rather than drawn from a closed colour set. This Part does not re-derive the operad reading of a manifest; it uses the manifest as the wiring of one installed application and adds only the registry-level composition across applications.

From Part III [12] it imports the port-type alphabet, the wiring operad $\mathcal{W}$, the sub-operad of tree wirings, and the statement that $\mathcal{W}$ and $\mathcal{O}$ are distinct operads with different concrete colour sets, related in one direction by type erasure. This Part defines no wiring of its own. Where it needs a harness-level wiring it takes a general wiring in Part III’s operad $\mathcal{W}_\top$ whose inner boxes are the stages of an installed application manifest. Such wirings are not tree wirings, for the reason given in Theorem 3.2.

Two dependencies run backwards relative to presentation order and are stated as such. Part II cites this Part for the notion of a certificate, because AgentHero attaches a policy envelope to individual operations and that envelope is Know-level. The stage-indexed presentation of Know in Theorem 3.5 exists so that Part II can cite it. Part III cites this Part for the same reason, when arguing that CatDB’s policy decisions and field provenance records are certificate-shaped without being separately addressable. Nothing in Parts I to III depends on any theorem proved here.

The results this Part supplies to the synthesis are Theorems 3.32, 6.3 and 6.5, together with the observation that the harness’s certificates are receipt-shaped. The synthesis needs the laxness result to weigh whether treating an agent as a lax monoidal functor adds anything to a purely economic account of why externalization happens.

2 Background

2.1 The harness pillar

Zhou et al. [19] organize the externalization of agent capability into four pillars: memory, skills, protocols, and harness engineering. The first three name things that are moved out of the model’s weights and into addressable structure. The fourth names the structure that holds the other three. Meng et al. [9] give an enumerative account of the same layer, decomposing a harness into environment, tools, context, security, learning and evaluation components. Pan et al. [18] study harnesses whose interfaces are natural language rather than typed calls and observe that harnesses behave as portable objects that can be moved between models.

Enumerations of this kind answer the question of what is in a harness. They do not answer the question of what a harness is, in the sense that would let one say when two harnesses are the same. Nor do they say what happens when two are composed, or which of their properties survive being recompiled onto a different runtime. Banu’s contribution [2] is to propose that these questions have standard answers once a harness is presented as an Architecture triple.

2.2 The Architecture triple

Banu’s presentation of the ArchAgents triple, which he takes by citation from de los Riscos, Corbacho and Arbib [4], runs as follows. An architecture is $\mathcal{A} = (G_{\mathcal{A}}, \text{Know}_{\mathcal{A}}, \Phi_{\mathcal{A}})$. Here $G_{\mathcal{A}}$ is syntactic wiring given as a graph of modules, ports and directed edges, and $\text{Know}_{\mathcal{A}}$ is a knowledge structure holding structural properties, invariants and certificates. The third component $\Phi_{\mathcal{A}}$ is a deployment map from abstract capability slots to concrete model or tool implementations. A morphism $f: \mathcal{A} \rightarrow \mathcal{B}$ is a structure-preserving translation, which Banu reads operationally as a compiler.

Two of Banu’s definitions are used here directly.

Definition 1 (Structural Guarantee as Certificate). A certificate is a triple $(\tau, \sigma, evds)$ where τ is a theorem statement, σ maps theorem symbols to architecture parameters, and $evds$ is a derivation that can be mechanically replayed to verify τ holds.

Definition 2 (Deployment Map). $\Phi: \text{Stages} \rightarrow \text{Models}$ assigns each stage a model tier. This map is a parameter of the architecture, not a fixed constant; different deployments can use different Φ while preserving the same (G, Know) .

The second definition has a consequence that shapes everything below. If Φ is a parameter, a morphism of architectures cannot be required to preserve it. Banu’s own compiler checks reflect this: of his three checks, graph preservation and certificate replay are equalities, while deployment-map preservation asks only that source stage names appear in the target, with mode mapping allowed to differ. We build that asymmetry into Theorem 3.9 and then show in Theorem 3.32 that it has a cost, because some checks a real harness performs read Φ .

On certificate preservation Banu is explicit that he is using an imported result rather than proving one:

ArchAgents state certificate preservation as Proposition 5.1. In this paper we use the result operationally, not as a new proof: a compiler that claims preservation must carry each source certificate’s theorem, parameters, and replayable evidence into the target Know structure. Functor laws alone are not sufficient for that claim, so our compiler checks certificate identity and verifier replay explicitly.

The same discipline is followed here, with one further step. The harness studied below has no theorem field, so $evds$ is replaced by an explicit pair of a witness and a decidable checker, and preservation is defined as stability of the checker under transport. This is weaker than Banu’s reading and strictly weaker than verification. It is also checkable by reading source, which the stronger readings are not.

2.3 Monoidal structure

The definitions are standard [8]. A monoidal category is a category $\mathcal{C}$ with a functor $\otimes: \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}$, a unit object I , and natural isomorphisms $a_{X,Y,Z}: (X \otimes Y) \otimes Z \rightarrow X \otimes (Y \otimes Z)$, $\lambda_X: I \otimes X \rightarrow X$ and $\rho_X: X \otimes I \rightarrow X$ satisfying the pentagon and triangle coherence conditions. It is symmetric when it further carries $s_{X,Y}: X \otimes Y \rightarrow Y \otimes X$ with $s_{Y,X}s_{X,Y} = \text{id}$, compatible with a and λ .

A lax monoidal functor $(A, \mu, \eta): (\mathcal{C}, \otimes, I) \rightarrow (\mathcal{D}, \boxtimes, J)$ is a functor A together with a natural transformation $\mu_{X,Y}: AX \boxtimes AY \rightarrow A(X \otimes Y)$ and a morphism $\eta: J \rightarrow AI$. The associativity and

unit diagrams commute:

$$\begin{array}{ccc}
(AX \boxtimes AY) \boxtimes AZ & \xrightarrow{a^D} & AX \boxtimes (AY \boxtimes AZ) \\
\downarrow \mu \boxtimes \text{id} & & \downarrow \text{id} \boxtimes \mu \\
A(X \otimes Y) \boxtimes AZ & & AX \boxtimes A(Y \otimes Z) \\
\downarrow \mu & & \downarrow \mu \\
A((X \otimes Y) \otimes Z) & \xrightarrow{Aa^C} & A(X \otimes (Y \otimes Z))
\end{array}$$

The functor is strong when every $\mu_{X,Y}$ and η is an isomorphism, and strict when they are identities. The distinction is the entire content of Theorem 6.3: laxness is not a technicality here but a statement that two applications installed in one harness cannot be separated again.

For the operadic and coalgebraic vocabulary the series uses elsewhere we cite the standard sources rather than restating them. The sources for Part III are Spivak’s operad of wiring diagrams [15] with its directed [13] and open-dynamical-system refinements [17]. Fong and Spivak [5] give the general compositional vocabulary, and Rutten [14] and Jacobs [6] Part I’s coalgebras.

3 The formal model

Throughout, T is a fixed port-type alphabet in the sense of Part III. The set R is a fixed nonempty set of deployment records. A record identifies the concrete realizer of a stage—a model endpoint, adapter process, or plugin binary—and any authorized workspace or artifact-store roots available to that stage.

3.1 Harness wirings

Part III owns the definition of typed ports, boxes and the wiring operad $\mathcal{W}_{\mathsf{T}}$, together with its sub-operad of tree wirings. This Part does not redefine any of them. It needs only a name for the particular wirings that occur at the harness level.

Definition 3.1 (Harness wiring). A *harness wiring* over T is a finite disjoint union of directed acyclic wiring diagrams in Part III’s operad $\mathcal{W}_{\mathsf{T}}$. Each summand has the stages and boundary declared by one installed application manifest. A one-application wiring is the special case with one summand. We write G for such a wiring and $\text{St}(G)$ for its finite set of inner boxes, called *stages*.

Remark 3.2 (Harness wirings are not tree wirings). Part III isolates the sub-operad of tree wirings, those diagrams that are linear (no supply port receives two demands) and rooted (each inner box supplies a single consumer, and the outer box has exactly one output port). It reports that this is the fragment `CatDB`’s query plans realize. Harness wirings are not in that fragment. The manifest edge type is declared as a pair of `OneOrMany` endpoint lists (`crates/dag-runtime/src/lib.rs`, lines 636 to 646), so one stage may supply several consumers, which breaks linearity. A manifest may also declare several terminal stages, which breaks rootedness. The harness therefore realizes a strictly larger fragment of $\mathcal{W}_{\mathsf{T}}$ than the protocol layer does, and the price is that Part III’s confinement argument, which is an induction over a tree, does not transfer.

Two further things are deliberately not claimed. Theorem 3.1 does not claim that Part III’s alphabet T and `AgentHero`’s string-keyed port names are the same alphabet. Part III’s type-erasure result gives the obstruction, and we return to it in Section 7. It does not claim that G determines an

operadic composite. Part II shows that one operadic operation requires an additional rooted output and unique supplier map, and that a multi-output invocation is split into coordinate generators. A PROP preserves the invocation as one operation. Where the results below need G they need only its stage set and its edges, never the composite, so this narrowing does not propagate into them.

3.2 Certificates

Definition 3.3 (Certificate, after Banu [2]). Fix a *sorted* value set, that is, a set V presented as a disjoint union $V = V_{\text{nm}} \sqcup V_{\text{pl}}$ of *name values*, which are stage and port identifiers, and *payload values*, which are everything else. For a wiring G write $\text{Nm}(G) \subseteq V_{\text{nm}}$ for the set of stage and port identifiers of G . Fix a set L of *field labels* and a set P of *parameter names*. A *witness over G* is a record, that is, a finite partial function $w: L \rightarrow \text{Nm}(G) \sqcup V_{\text{pl}}$. We write $w\ell$ for $w(\ell)$, call $\text{dom}(w)$ its *shape*, and write W_G for the set of witnesses over G . Fix a set $\mathcal{T}$ of *claim identifiers* together with a function $\text{par}: \mathcal{T} \rightarrow \mathcal{P}_{\text{fin}}(P)$ assigning to each claim identifier its finite set of parameter names. A *certificate over G* is a quadruple

$$c = (\tau_c, \sigma_c, \text{scope}(c), \text{chk}_c)$$

where $\tau_c \in \mathcal{T}$; $\sigma_c: \text{par}(\tau_c) \rightarrow V$ is a partial function; $\text{scope}(c) \subseteq \text{St}(G)$ is the set of stages the certificate is about; and $\text{chk}_c: R^{\text{St}(G)} \times W_G \rightarrow \{\top, \perp\}$ is a decidable predicate, the *replay check*. The first argument permits a check to consult deployment data. Once an architecture $\mathcal{A}$ is fixed, write $\text{chk}_c^{\mathcal{A}}(w) = \text{chk}_c(\Phi_{\mathcal{A}}, w)$. A witness w *satisfies* c in $\mathcal{A}$ when $\text{chk}_c^{\mathcal{A}}(w) = \top$. We write $\text{Cert}(G)$ for the set of certificates over G .

Remark 3.4 (The sorting hypothesis and its status in the code). The disjointness of V_{nm} from V_{pl} is a hypothesis, and it is the hypothesis Theorem 3.32(1) needs. Without it a translation could send a stage identifier onto a payload string that was already present, and transport would stop being injective. The hypothesis is not automatic: as stated in Part II, **AgentHero**’s manifest ports are string-keyed with JSON payloads, so at the manifest level names and payloads inhabit one untyped space. It is satisfied by the certificates this paper is about. **HookRequest** and **HookReceipt** (`crates/orchestrator/src/hooks.rs`, lines 404 and 430) declare `plugin`, `hook_key` and `idempotency_key` as separate `String` fields, `event_id` as an `Option<i64>`, and the plugin-defined payload as a single `output` field of `JSON` type. A name field and a payload field are therefore distinct record positions and cannot be confused. Where the hypothesis fails, so does the theorem, and Section 8 records that this is one more thing the untyped port vocabulary costs.

Theorem 3.3 is Banu’s Definition 1 with two changes, both forced by what implementations carry. The evidence component *evds* is split into a witness, which is data an implementation retains, and a checker, which is code an implementation runs. Banu’s phrase “a derivation that can be mechanically replayed” is exactly that pair. A scope is also added. Banu’s certificates are architecture-wide. Real harnesses attach policy to individual stages as well as to the whole system, and the scope records which.

Remark 3.5 (Know as a stage-indexed family). Given a finite set Know of certificates over G , define $\text{Know}_n = \{c \in \text{Know} : n \in \text{scope}(c)\}$ for each $n \in \text{St}(G)$. The family $(\text{Know}_n)_{n \in \text{St}(G)}$ together with the certificates of empty scope determines Know , and conversely, so the two presentations are interchangeable. Part II cites the family presentation, because **AgentHero**’s per-operation policy envelopes are naturally indexed by stage.

Definition 3.6 (Receipt-shaped certificate). A certificate c is *receipt-shaped* when τ_c is a schema identifier carrying no logical content, σ_c is a tuple of identity fields, and $\text{chk}_c(\Phi, w)$ is, for every Φ ,

the same conjunction of finitely many equalities $w\ell_i = \sigma_c(p_i)$, for pairs $(\ell_i, p_i) \in L \times \text{par}(\tau_c)$ fixed by τ_c , together with finitely many syntactic well-formedness tests on w . A syntactic test is one that depends only on the shape of w and on the sort and syntactic type of each field, never on a payload value.

A receipt-shaped certificate says that a named party produced a well-formed record with the expected identity, and nothing else. It carries no theorem. The following makes “and nothing else” precise rather than leaving it as an admission.

Proposition 3.7 (What a receipt-shaped check can see). *Let c be receipt-shaped and let π be any type-preserving bijection of the payload values V_{pl} fixing the values appearing in σ_c . It acts on witnesses by replacing each payload field v by $\pi(v)$ and leaving name fields and record shape alone. Then $\text{chk}_c(\Phi, \pi \cdot w) = \text{chk}_c(\Phi, w)$ for every Φ and witness w . Consequently no receipt-shaped certificate distinguishes two witnesses that agree on their identity fields and on their record shape, however their contents differ.*

Proof. By Theorem 3.6, $\text{chk}_c(\Phi, w)$ is a conjunction of equalities $w\ell_i = \sigma_c(p_i)$ and of syntactic well-formedness tests on w . Each equality compares a field against a value of σ_c . If that field is a name field then π does not touch it. If it is a payload field then both sides are fixed by π , the right side by hypothesis. If the equality holds, both sides remain equal; if it fails, injectivity of π prevents unequal payloads from becoming equal. The well-formedness tests depend on record shape and field types, which π preserves. So each conjunct has the same truth value at w and at $\pi \cdot w$. $\square$

Theorem 3.7 is the exact sense in which the harness’s knowledge layer is knowledge-free. It detects identity and shape and is blind to content by construction, so any claim of the form “this certificate rules out behaviour B ” is unavailable unless B is a statement about identity or shape. Section 8 records that every certificate found in the harness is receipt-shaped, and this proposition says what that costs.

3.3 Architectures and their morphisms

Definition 3.8 (Architecture, after Banu [2]). An *architecture* is a triple $\mathcal{A} = (G_{\mathcal{A}}, \text{Know}_{\mathcal{A}}, \Phi_{\mathcal{A}})$ where $G_{\mathcal{A}}$ is a harness wiring, $\text{Know}_{\mathcal{A}}$ is a finite set of certificates over $G_{\mathcal{A}}$, and $\Phi_{\mathcal{A}}: \text{St}(G_{\mathcal{A}}) \rightarrow R$ is a function, the *deployment map*. We write $\mathcal{H}$ for an architecture presented as a harness.

Definition 3.9 (Architecture morphism). A morphism $f: \mathcal{A} \rightarrow \mathcal{B}$ includes an injection

$$f_{\text{St}}: \text{St}(G_{\mathcal{A}}) \hookrightarrow \text{St}(G_{\mathcal{B}})$$

that extends to a port-type-preserving embedding of $G_{\mathcal{A}}$ into $G_{\mathcal{B}}$ as a sub-wiring, together with a function $f_{\text{Know}}: \text{Know}_{\mathcal{A}} \rightarrow \text{Know}_{\mathcal{B}}$ such that for every $c \in \text{Know}_{\mathcal{A}}$, writing $c' = f_{\text{Know}}(c)$:

- (M1) $\tau_{c'} = \tau_c$ (claim identity);
- (M2) $\sigma_{c'} = f_* \circ \sigma_c$, where $f_*: \text{Nm}(G_{\mathcal{A}}) \sqcup V_{\text{pl}} \rightarrow \text{Nm}(G_{\mathcal{B}}) \sqcup V_{\text{pl}}$ is the stage and port renaming induced by the sub-wiring embedding on $\text{Nm}(G_{\mathcal{A}})$ and the identity on V_{pl} (parameter transport); f_* is injective because the embedding is;
- (M3) $\text{scope}(c') = f_{\text{St}}(\text{scope}(c))$ (scope transport);
- (M4) for every witness $w \in W_{G_{\mathcal{A}}}$, if $\text{chk}_c^{\mathcal{A}}(w) = \top$ then $\text{chk}_{c'}^{\mathcal{B}}(f_{\#}w) = \top$, where $f_{\#}: W_{G_{\mathcal{A}}} \rightarrow W_{G_{\mathcal{B}}}$ is post-composition with f_* , that is, $f_{\#}w = f_* \circ w$, a witness of the same shape whose values are transported (check acceptance).

No condition is imposed on Φ , in accordance with Banu's Definition 2. Both transports are functorial in the stage map: $(gf)_* = g_*f_*$ because renamings compose, and hence $(gf)_\# = g_\#f_\#$.

Proposition 3.10. *Architectures and their morphisms form a category **Arch**, with composition given by composing the underlying stage injections and certificate functions, and identities given by identity maps.*

Proof. Identities satisfy (M1) to (M4) because id_* and $\text{id}_\#$ are identities. For composition, let $f: \mathcal{A} \rightarrow \mathcal{B}$ and $g: \mathcal{B} \rightarrow \mathcal{C}$. The composite stage map $g_{\text{St}}f_{\text{St}}$ is an injection and the composite of two sub-wiring embeddings preserving port types is one. Set $(gf)_{\text{Know}} = g_{\text{Know}}f_{\text{Know}}$. Then (M1) holds since $\tau_{g_{\text{Know}}f_{\text{Know}}(c)} = \tau_{f_{\text{Know}}(c)} = \tau_c$. For (M2) and (M3), transport is functorial in the stage map: $(gf)_* = g_*f_*$ and $(gf)_{\text{St}} = g_{\text{St}}f_{\text{St}}$, so $\sigma_{(gf)_{\text{Know}}(c)} = g_*f_*\sigma_c = (gf)_*\sigma_c$ and likewise for scopes. For (M4), suppose $\text{chk}_c^{\mathcal{A}}(w) = \top$. Applying (M4) for f gives $\text{chk}_{f_{\text{Know}}(c)}^{\mathcal{B}}(f_\#w) = \top$; applying it for g at the witness $f_\#w$ gives $\text{chk}_{g_{\text{Know}}f_{\text{Know}}(c)}^{\mathcal{C}}(g_\#f_\#w) = \top$, and $g_\#f_\# = (gf)_\#$ because $f_\#$ is post-composition with f_* and $(gf)_* = g_*f_*$. Associativity and unitality follow from those of function composition. $\square$

Proposition 3.11 (Forgetting deployment on independent checks). *Let $\mathbf{Arch}^{\text{ind}}$ be the full subcategory in which every checker factors through (G, Know) as in Section 3.7. Let $\mathbf{Arch}_0^{\text{ind}}$ forget Φ from those objects and use (M1) to (M4) with the deployment-independent checker. Then the forgetful functor $U: \mathbf{Arch}^{\text{ind}} \rightarrow \mathbf{Arch}_0^{\text{ind}}$ is full, faithful, and surjective on objects, hence an equivalence.*

Proof. On this subcategory, (M4) is independent of both deployment maps. The hom-sets before and after forgetting Φ are therefore identical pairs $(f_{\text{St}}, f_{\text{Know}})$. Since R is nonempty, every pair (G, Know) admits at least one deployment map. Thus U is full, faithful, and surjective on objects. $\square$

Remark 3.12 (Scope of deployment forgetting). The equivalence does not extend to deployment-reading checks, because (M4) then depends on the source and target deployment maps. This is the separation used below: deployment is removable on the independent-check fragment and structural on the remainder. **Sys** retains it as component labelling in both cases.

Definition 3.13 (The deployment-preserving refinement). Let $\mathbf{Arch}^\Phi$ have the same objects as **Arch** and, as morphisms $\mathcal{A} \rightarrow \mathcal{B}$, those morphisms f of **Arch** that additionally satisfy $\Phi_{\mathcal{B}} \circ f_{\text{St}} = \Phi_{\mathcal{A}}$.

Proposition 3.14. $\mathbf{Arch}^\Phi$ is a subcategory of **Arch** containing all objects, and it is not equivalent to **Arch** under the identity on objects: there are architectures $\mathcal{A}, \mathcal{B}$ with $\text{Hom}_{\mathbf{Arch}}(\mathcal{A}, \mathcal{B})$ nonempty and $\text{Hom}_{\mathbf{Arch}^\Phi}(\mathcal{A}, \mathcal{B})$ empty.

Proof. Closure under composition and identities is immediate from $\Phi_{\mathcal{C}}g_{\text{St}}f_{\text{St}} = \Phi_{\mathcal{B}}f_{\text{St}} = \Phi_{\mathcal{A}}$. For the separation, take $G_{\mathcal{A}} = G_{\mathcal{B}}$ with one stage n , empty certificate sets, and $\Phi_{\mathcal{A}}(n) \neq \Phi_{\mathcal{B}}(n)$. The identity on stages is a morphism of **Arch** and satisfies no equation of Theorem 3.13, and it is the only candidate, so the $\mathbf{Arch}^\Phi$ hom-set is empty. $\square$

Remark 3.15. In $\mathbf{Arch}^\Phi$ the deployment-changing identity used in Theorem 3.32(2) is not a morphism. Checks that depend only on deployment values at transported source stages are therefore stable under that counterexample. This does not cover an arbitrary target check: it may inspect new target stages outside the image of f_{St} . The two categories still make different trade-offs. **Arch** is Banu's setting and permits redeployment; $\mathbf{Arch}^\Phi$ forbids changes on transported stages. A realization cannot both permit such a change and treat its old deployment-relative verdict as invariant.

Remark 3.16. Condition (M4) is one-directional. It says the target check is at least as permissive as the source check on transported witnesses, which is what “the certificate still holds after translation” should mean. The converse would say that the target check is no stronger than the source check on the same certificate. That is a genuine additional property rather than part of the definition. A target runtime may validate more fields of the same receipt than the source did, and such a translation is one a compiler is entitled to make. Building the equivalence into the definition would make that translation not a morphism and would leave nothing to study. We therefore keep the implication in (M4) and study the equivalence separately, under the name replay stability (Theorem 3.30).

3.4 The monoidal product

Definition 3.17 (Tensor of architectures). For architectures $\mathcal{A}, \mathcal{B}$ set

$$\mathcal{A} \otimes \mathcal{B} = (G_{\mathcal{A}} \sqcup G_{\mathcal{B}}, \text{Know}_{\mathcal{A}} \sqcup \text{Know}_{\mathcal{B}}, [\Phi_{\mathcal{A}}, \Phi_{\mathcal{B}}]),$$

where $G_{\mathcal{A}} \sqcup G_{\mathcal{B}}$ is the wiring whose stages are the tagged disjoint union $\text{St}(G_{\mathcal{A}}) \sqcup \text{St}(G_{\mathcal{B}})$, whose wires are those of $G_{\mathcal{A}}$ and of $G_{\mathcal{B}}$ with no wire between the summands, and whose outer box is the disjoint union of the two outer boxes. A certificate from either summand keeps its claim identifier, transports its parameter values and scope along the tagging injection, and evaluates its checker by restricting the deployment map to that summand and untagging witness names. The copairing $[\Phi_{\mathcal{A}}, \Phi_{\mathcal{B}}]$ is the function out of the disjoint union that restricts to $\Phi_{\mathcal{A}}$ on the first summand and to $\Phi_{\mathcal{B}}$ on the second. Square brackets denote copairings out of a disjoint union throughout. The unit I is the architecture with no stages, no certificates and the empty deployment map.

Proposition 3.18. *(Arch, $\otimes, I$) is a symmetric monoidal category.*

Proof. Functoriality of $\otimes$: given $f: \mathcal{A} \rightarrow \mathcal{A}'$ and $g: \mathcal{B} \rightarrow \mathcal{B}'$, the map $f \sqcup g$ on tagged stages is an injection, embeds $G_{\mathcal{A}} \sqcup G_{\mathcal{B}}$ as a sub-wiring, and satisfies (M1) to (M4) summand-wise. Each condition is stated pointwise on certificates, and certificates in the tensor lie in exactly one summand. Preservation of identities and composites is immediate.

Coherence: tagged disjoint union has canonical associator, unitor, and symmetry bijections. Direct evaluation on tagged elements verifies the pentagon, triangle, and symmetry equations. Each such bijection lifts uniquely to a wiring isomorphism, because there are no wires across summands and each summand’s wiring is carried identically. On certificates the induced maps are the corresponding re-taggings, which preserve τ , transport σ and scope along a bijection, and transport chk by the restriction-and-untagging rule above. So (M1) to (M4) hold with equality in (M4), and the inverses are morphisms as well. On deployment maps the copairing satisfies $[[\Phi_{\mathcal{A}}, \Phi_{\mathcal{B}}], \Phi_{\mathcal{C}}] = [\Phi_{\mathcal{A}}, [\Phi_{\mathcal{B}}, \Phi_{\mathcal{C}}]]$ modulo the same bijection, and no morphism condition constrains Φ in any case. The unit conditions hold because I has empty stage set. $\square$

Remark 3.19. $\otimes$ places no wires between the summands, so it models two harnesses installed side by side, not two harnesses connected to each other. Connected composition is operadic substitution, which is Part II’s and Part III’s subject. The distinction matters for Theorem 6.3. What fails to be recoverable from the composite is not information about how the two applications talk to each other, since they do not. It is information about the order in which the shared runtime admitted their work.

A real harness does not tag its stage names. It installs applications into one namespace and refuses collisions. That gives a partial version of $\otimes$ which is what the implementation realizes.

Definition 3.20 (Globally named architecture). A *globally named* architecture is an architecture together with an injection $\nu: \text{St}(G) \hookrightarrow N$ into a fixed set N of global names. Two globally named architectures are *compatible* when the images of their name maps are disjoint. For compatible $\mathcal{A}, \mathcal{B}$, the *untagged tensor* $\mathcal{A} \otimes_N \mathcal{B}$ is $\mathcal{A} \otimes \mathcal{B}$ with name map $[\nu_{\mathcal{A}}, \nu_{\mathcal{B}}]$, which is injective exactly by compatibility.

Proposition 3.21. *The untagged tensor $\otimes_N$ is a partial binary operation on globally named architectures, defined exactly on compatible pairs, and the forgetful map to **Arch** sends $\mathcal{A} \otimes_N \mathcal{B}$ to $\mathcal{A} \otimes \mathcal{B}$. Compatibility is decidable in time $O(n \log n)$ in the total number n of global names, using an ordered map.*

Proof. The copairing $[\nu_{\mathcal{A}}, \nu_{\mathcal{B}}]$ of two injections out of a coproduct is injective if and only if their images are disjoint, which is the definition of compatibility, so $\otimes_N$ is defined exactly there. Forgetting ν gives $\mathcal{A} \otimes \mathcal{B}$ by construction. For decidability, insert every name into an ordered map keyed by name with the owning architecture as value, and report a collision when a key is already present under a different owner. Each of the n insertions and lookups costs $O(\log n)$ in an ordered map, giving $O(n \log n)$ overall. We state the ordered-map bound rather than the average-case hash bound because the implementation examined in Section 5.6 uses an ordered map. $\square$

Section 6 identifies the implementation of this decision procedure and shows that it is exactly the check the registry performs before admitting a pair of installed applications.

3.5 Systems

The target category must be concrete enough that its objects correspond to something a harness declares, and abstract enough that a functor into it can be defined without appeal to observed runs. The second requirement rules out the obvious choice of carrying fence values directly. A fence value may be allocated before the guarded write commits. The actual fence carried by an actual receipt is therefore a fact about a run, and cannot appear in the image of a functor defined on architectures. What a store design does fix, statically and readably from its schema, is which receipts are drawn from a common monotone sequence. That is the invariant **Sys** records.

Definition 3.22 (Sequencing relation). A *sequencing relation* on a set Rcpt is an equivalence relation $\sim$ on Rcpt . Two receipts are *co-sequenced* when they are $\sim$ -related, meaning that the store design draws their fence values from one monotone sequence and therefore orders them against each other. Receipts in different classes are ordered by no common sequence.

Remark 3.23. **Sys** deliberately records the partition into sequences and not the positions within a sequence. Positions are assigned at sequence-allocation time and differ between runs. The partition is determined by the schema: a column drawn from one database sequence co-sequences every row that reads it, and a per-row counter co-sequences only the attempts on that row. Every claim made below about **Sys** is therefore a claim about a schema, checkable by reading a migration file, and none is a claim about a trace.

Definition 3.24 (The category **Sys**). An object of **Sys** is a quintuple

$$X = (\text{Comp}_X, \lambda_X, \text{Rcpt}_X, \text{fld}_X, \sim_X)$$

in which Comp_X is a finite set of named components. The *deployment labelling* $\lambda_X: \text{Comp}_X \rightarrow R$ assigns each component its deployment record. The set Rcpt_X holds *receipt identities*, each a record carrying a schema field and an identity field, read off by a function $\text{fld}_X: \text{Rcpt}_X \rightarrow \Sigma$ into

a fixed set Σ of schema and identity values. The last component $\sim_X$ is a sequencing relation on Rcpt_X . A morphism $h: X \rightarrow Y$ is a pair $(h_{\text{Comp}}, h_{\text{Rcpt}})$ with $h_{\text{Comp}}: \text{Comp}_X \hookrightarrow \text{Comp}_Y$ an injection, $h_{\text{Rcpt}}: \text{Rcpt}_X \rightarrow \text{Rcpt}_Y$ a function with $\text{fld}_Y \circ h_{\text{Rcpt}} = \text{fld}_X$, so that the schema and identity fields are preserved, and

$$r \sim_X r' \implies h_{\text{Rcpt}}(r) \sim_Y h_{\text{Rcpt}}(r').$$

No condition is imposed on the labelling, in deliberate parallel with Theorem 3.9 and Theorem 3.11: a translation may redeploy a component. Composition is componentwise. The monoidal product is

$$X \boxtimes Y = (\text{Comp}_X \sqcup \text{Comp}_Y, [\lambda_X, \lambda_Y], \text{Rcpt}_X \sqcup \text{Rcpt}_Y, [\text{fld}_X, \text{fld}_Y], \sim_X \sqcup \sim_Y),$$

where $[\lambda_X, \lambda_Y]$ is the copairing, the field map of the product is the copairing $[\text{fld}_X, \text{fld}_Y]$, and $\sim_X \sqcup \sim_Y$ relates two receipt identities only when they lie in the same summand and are related there. The unit J has empty component and receipt sets and the unique empty labelling, field map, and relation.

Receipt maps are not required to be injective. Injectivity is imposed on components, where it says that distinct named components stay distinct. It is not imposed on receipt identities, where nothing in the intended reading forbids a translation from identifying two shapes that carry the same schema and identity fields.

Proposition 3.25. *($\mathbf{Sys}, \boxtimes, J$) is a symmetric monoidal category.*

Proof. $\mathbf{Sys}$ is a category: identities satisfy the sequencing implication trivially, and if h and k both satisfy it then so does kh , since $r \sim_X r'$ gives $h(r) \sim_Y h(r')$ gives $kh(r) \sim_Z kh(r')$; injections compose. Tagged disjoint union has its standard symmetric monoidal associator, unitors, and symmetry. Each canonical map is a bijection whose inverse also preserves the relation, because on a coproduct the relation is determined summand-wise and the canonical bijections respect summands; hence each is an isomorphism of $\mathbf{Sys}$. Pentagon, triangle and symmetry conditions hold because they hold for coproducts of sets and every component of each diagram is determined by its underlying function. $\square$

The receipt component of Theorem 3.24 is a standard structure under a different name.

Proposition 3.26 ($\mathbf{Sys}$ is a product of two standard constructions). *Let $\mathbf{Eq}$ be the category of pairs (set, equivalence relation) with relation-preserving functions. Let $\mathbf{Set}^{\rightarrow}$ be the full subcategory of the arrow category $\mathbf{Set}^2$ on the surjective functions, with morphisms the commuting squares. Let ∇_Σ be the total relation on Σ and write $\mathbf{Eq}/\Sigma$ for the slice category $\mathbf{Eq}/(\Sigma, \nabla_\Sigma)$. Its objects are triples $(\text{Rcpt}, \sim, \text{fld}: \text{Rcpt} \rightarrow \Sigma)$ and its morphisms are relation-preserving functions h with $\text{fld}' \circ h = \text{fld}$. Then:*

1. *sending $(\text{Rcpt}, \sim)$ to the quotient map $q: \text{Rcpt} \twoheadrightarrow \text{Rcpt}/\sim$ is an equivalence $\mathbf{Eq} \simeq \mathbf{Set}^{\rightarrow}$ carrying $\sqcup$ to the coproduct;*
2. *writing $\mathbf{Lab}$ for the category of finite R -labelled sets with injections that need not preserve labels, there is an isomorphism of categories $\mathbf{Sys} \cong \mathbf{Lab} \times \mathbf{Eq}/\Sigma$, and it carries $\boxtimes$ to the componentwise disjoint-union monoidal product.*

Proof. (1) A function $h: \text{Rcpt} \rightarrow \text{Rcpt}'$ satisfies $r \sim r' \implies h(r) \sim' h(r')$ if and only if there is a necessarily unique $\bar{h}$ with $\bar{h} \circ q = q' \circ h$, since q is surjective and the condition says exactly that $q'h$ is constant on $\sim$ -classes. The assignment is therefore full and faithful, and it is essentially

surjective because every surjection $q: R \rightarrow B$ is the quotient by the equivalence relation “same fibre”. Coproducts of surjections are computed componentwise, which is the disjoint sum of the relations. (2) Every function into Σ preserves the total relation ∇_Σ , so an object of the slice $\mathbf{Eq}/\Sigma$ is exactly a set with an equivalence relation and an unconstrained field map, and a morphism of the slice is exactly a relation-preserving function commuting with the field maps. The slice therefore imposes the field-preservation condition of Theorem 3.24 and nothing else. By Theorem 3.24, an object of $\mathbf{Sys}$ is then a pair consisting of an object of $\mathbf{Lab}$ and an object of $\mathbf{Eq}/\Sigma$, and a morphism is a pair consisting of a morphism of each, with no condition linking them. Composition and identities are componentwise. That is the definition of a product category. The product respects $\boxtimes$ because tagged disjoint union acts componentwise on labelled component sets and on receipt sets, relations, and field maps. No categorical coproduct in the injection category is asserted. $\square$

Remark 3.27. Theorem 3.26 answers the objection that $\mathbf{Sys}$ is a transcription of one database schema. It is a product of two standard constructions. The first is finite labelled sets with injections. The second is a slice of the category of sets with equivalence relations over the fixed set of schema and identity values, which by part (1) is the same as sets fibred over an index set and labelled by Σ . The only application-specific content is the reading of the two factors, components labelled by their realizers and receipts fibred over the sequence their fences come from. What is specific to harnesses is the interpretation, not the target. The two factors are also exactly the two things Theorem 3.9 refuses to constrain and Theorem 3.8 does carry. So $\mathbf{Sys}$ is where the deployment data that $\mathbf{Arch}$ drops becomes visible again.

Remark 3.28. The sequencing relation is the modelling choice that carries the paper’s main negative result. A fence is a token by which a store can reject a writer whose claim has been superseded, and a fence is only meaningful relative to the sequence it is drawn from. Whether two writers’ fences are drawn from one sequence is a design decision recorded in a schema, not an accident of scheduling. The morphism condition is an implication and not an equivalence. A morphism may co-sequence receipts that were not co-sequenced before, which is what installing two applications into one store does, and it may not separate receipts that were. Theorem 6.3 is exactly the observation that this asymmetry is not reversible.

3.6 Interpretations

Definition 3.29 (Interpretation, after Banu [2]). An *interpretation* is a lax monoidal functor

$$(A, \mu, \eta): (\mathbf{Arch}, \otimes, I) \longrightarrow (\mathbf{Sys}, \boxtimes, J).$$

We call A *strong on a full monoidal subcategory* $\mathcal{F} \subseteq \mathbf{Arch}$ when $\mu_{X,Y}$ is an isomorphism for all $X, Y \in \mathcal{F}$.

Banu describes an agent as a monoidal functor interpreting an architecture in a concrete system. Theorem 3.29 makes the laxness explicit and, by Theorem 3.29’s relativization to a subcategory, makes room for one abstraction to be strong on a fragment and another to be lax. Section 6 exhibits both receipt-sequencing cases in $\mathbf{AgentHero}$; it does not construct a single operational interpretation covering both.

3.7 Replay stability

Definition 3.30 (Replay stability). Let $f: \mathcal{A} \rightarrow \mathcal{B}$ be a morphism and $c \in \text{Know}_{\mathcal{A}}$. The certificate c is *replay-stable under* f when for every witness $w \in W_{G_{\mathcal{A}}}$,

$$\text{chk}_{f_{\text{Know}(c)}}^{\mathcal{B}}(f_{\#}w) = \text{chk}_c^{\mathcal{A}}(w).$$

It is *replay-stable* when it is replay-stable under every morphism out of $\mathcal{A}$ that is defined on it.

Replay stability strengthens (M4) from an implication to an equivalence. It is the formal content of “certificate preservation” here: the check neither loosens nor tightens under translation. It is not verification. A replay-stable certificate whose check is trivially true is replay-stable and says nothing.

The relation between the two conditions is exact, and it is not a change of subject.

Proposition 3.31 (Replay stability is a wide subcategory). *Let $\mathbf{Arch}^=$ have the same objects as $\mathbf{Arch}$ and, as morphisms, those f for which (M4) holds with equality for every certificate. Then $\mathbf{Arch}^=$ is a wide subcategory of $\mathbf{Arch}$. For a morphism $f: \mathcal{A} \rightarrow \mathcal{B}$ of $\mathbf{Arch}$ the following are equivalent: every certificate of $\mathcal{A}$ is replay-stable under f ; and f lies in $\mathbf{Arch}^=$.*

Proof. Identities satisfy equality. If f and g satisfy equality then

$$\begin{aligned} \text{chk}_{(gf)_{\text{Know}(c)}}^C((gf)_{\#}w) &= \text{chk}_{g_{\text{Know}(f_{\text{Know}(c)})}}^C(g_{\#}f_{\#}w) \\ &= \text{chk}_{f_{\text{Know}(c)}}^B(f_{\#}w) = \text{chk}_c^A(w). \end{aligned}$$

The second equality uses stability for g at $f_{\text{Know}(c)}$ and $f_{\#}w$; the third uses stability for f . So $\mathbf{Arch}^=$ is closed under composition and contains every object. The stated equivalence is Theorem 3.30 read certificate by certificate. $\square$

So the object of study is not an ad-hoc predicate attached to a category with the wrong morphisms. It is the wide subcategory $\mathbf{Arch}^= \subseteq \mathbf{Arch}$, and the question “which certificates are preserved” is the question “which architectures have the property that every morphism out of them lands in $\mathbf{Arch}^=$ ”. Theorem 3.32 answers it.

We now isolate the property that decides replay stability. Say that a check chk_c *factors through* (G, Know) when there is a function k such that $\text{chk}_c(\Phi, w) = k(\tau_c, \sigma_c, \text{scope}(c), w)$ for every Φ , that is, when the check reads only the certificate’s own data and the witness. Say that it *reads* Φ when its value depends on $\Phi_{\mathcal{A}}$ with the other arguments held fixed.

Theorem 3.32 (Separation of preservation). *Let $\mathcal{A}$ be an architecture and $c \in \text{Know}_{\mathcal{A}}$.*

1. *If chk_c factors through (G, Know) and c is receipt-shaped, then c is replay-stable under every morphism out of $\mathcal{A}$.*
2. *There exist an architecture $\mathcal{A}$, a certificate $c \in \text{Know}_{\mathcal{A}}$ whose check reads Φ , and a morphism $f: \mathcal{A} \rightarrow \mathcal{B}$ under which c is not replay-stable. Indeed f may be taken to be the identity on (G, Know) , changing only Φ .*

Proof. (1) Let $f: \mathcal{A} \rightarrow \mathcal{B}$ and $c' = f_{\text{Know}(c)}$. Since c is receipt-shaped, $\text{chk}_c^A(w)$ is a conjunction of equalities $w\ell_i = \sigma_c(p_i)$ between named fields of w and parameter values, together with syntactic tests on w that do not mention σ_c . By (M1) the two certificates have the same claim identifier, hence the same field list (ℓ_i) and the same syntactic tests. By (M2), $\sigma_{c'} = f_*\sigma_c$. By Theorem 3.9, $f_{\#}w = f_* \circ w$, so $(f_{\#}w)\ell_i = f_*(w\ell_i)$ for each i . Since f_* is injective, $f_*(w\ell_i) = f_*(\sigma_c(p_i))$ if and only if $w\ell_i = \sigma_c(p_i)$. The syntactic tests are preserved and reflected because $f_{\#}$ preserves the shape of w and the sort and syntactic type of every field, and the tests depend on nothing else. Conjoining, the two checks agree on every witness.

(2) Take $\text{St}(G) = \{n\}$ and one certificate c with claim identifier “the witness names a path below the root assigned to n ”. Let $R = \{r_0, r_1\}$ be two realizers with distinct root directories, and set $\text{chk}_c(\Phi, w) = \top$ if and only if the payload field $w.\text{path}$ is a path below $\text{root}(\Phi(n))$. Let $\mathcal{B}$

have the same G and the same Know but $\Phi_{\mathcal{B}}(n) = r_1$ while $\Phi_{\mathcal{A}}(n) = r_0$. The identity on stages and certificates is a morphism $\mathcal{A} \rightarrow \mathcal{B}$ when $\text{root}(r_0) \subsetneq \text{root}(r_1)$: (M1) to (M3) are identities, and every path accepted under r_0 is accepted under r_1 , so (M4) holds. Yet $\text{chk}_c^{\mathcal{B}}$ accepts strictly more. Then a witness w whose *path* field lies below $\text{root}(r_1)$ but not below $\text{root}(r_0)$ has $\text{chk}_c^{\mathcal{A}}(w) = \perp$ and $\text{chk}_c^{\mathcal{B}}(f_{\#}w) = \top$, since $f_{\#}$ fixes payload fields. So equality fails and c is not replay-stable under f . $\square$

Remark 3.33. Theorem 3.32 says the price of Banu’s Definition 2 is paid by the certificates. Making Φ a free parameter is what allows one architecture to be deployed against different models. The theorem shows that this freedom can make a deployment-reading check unstable; it does not say that every such check is unstable. A harness that requires invariance must either keep a check on the (G, Know) side of the line or prove stability for the deployment data it reads. Section 6 exhibits one check on each side.

Proposition 3.34 (Admission is not preserved). *Define the admission predicate of an architecture at a witness w by $\text{Adm}_{\mathcal{A}}(w) = \bigwedge_{c \in \text{Know}_{\mathcal{A}}} \text{chk}_c^{\mathcal{A}}(w)$. There is a morphism $f: \mathcal{A} \rightarrow \mathcal{B}$ under which every certificate of $\mathcal{A}$ is replay-stable and yet $\text{Adm}_{\mathcal{A}}(w) = \top$ while $\text{Adm}_{\mathcal{B}}(f_{\#}w) = \perp$.*

Proof. Let $\mathcal{A}$ have a single stage and a single certificate c with $\text{chk}_c \equiv \top$. Let $\mathcal{B}$ have the same wiring and deployment map and $\text{Know}_{\mathcal{B}} = \{c, d\}$ with $\text{chk}_d \equiv \perp$. The map that is the identity on stages and sends $c \mapsto c$ is a morphism: (M1) to (M3) are identities and (M4) holds because chk_c accepts everything in both. Every certificate of $\mathcal{A}$, namely c , is replay-stable, with equality in Theorem 3.30. But $\text{Adm}_{\mathcal{A}}(w) = \text{chk}_c^{\mathcal{A}}(w) = \top$ and $\text{Adm}_{\mathcal{B}}(f_{\#}w) = \text{chk}_c^{\mathcal{B}}(f_{\#}w) \wedge \text{chk}_d^{\mathcal{B}}(f_{\#}w) = \perp$. $\square$

Remark 3.35. Adm is antitone in Know : adding certificates can only shrink the set of admitted witnesses. Certificate preservation is a statement about each certificate separately and is therefore silent about the conjunction. This is not a defect of ArchAgents’ Proposition 5.1, which is a statement about certificates. It is a limit on what an operational use of that proposition can conclude. A compiler that preserves every certificate can still turn an admitted operation into a refused one, and Section 6 shows that installing an additional pre-policy hook is precisely such a translation in the system studied here.

4 Harness laws

The definitions above are satisfied by many things, including useless ones. The following seven laws state what a realization must additionally do for the interpretation to carry weight. They are stated so that each can be settled by reading source at a fixed commit. Section 6 settles them for AgentHero.

Law 1 (Separability of Know from G). Each certificate is addressable independently of the wiring it scopes: there is a representation of Know that can be read, listed and modified without editing G .

Law 2 (Receipt identity replay). For each certificate the implementation performs a total, decidable check comparing the identity fields of a produced witness against the certificate’s parameters, and refuses the witness on any mismatch.

Law 3 (Deterministic replay identity). Each certificate carries an identity that is a deterministic function of transported data alone, so that two evaluations of the same certificate against the same event are recognizable as the same evaluation.

Law 4 (Fail-closed admission). If any certificate scoped to an operation fails its check, or if the check cannot be evaluated, the operation is refused rather than admitted.

Law 5 (Fence exclusion). An effect that has been superseded cannot be recorded: a writer holding a stale fence is refused by the store, and the refusal is detected rather than silently ignored.

Law 6 (Φ -independence of checks). No certificate’s check reads the deployment map. Equivalently, every check factors through (G, Know) in the sense of Section 3.7.

Law 7 (Name disjointness under $\otimes$). The implementation refuses to compose two architectures whose global stage names collide, so that the untagged tensor of Theorem 3.20 is defined whenever the implementation admits the composite.

Law 6 is the one that Theorem 3.32 shows to be load-bearing: a realization that violates it has certificates that do not survive redeployment. Law 4 and Theorem 3.34 pull in opposite directions, and a realization that satisfies Law 4 is thereby one whose admission predicate is not preserved by Know-extending morphisms. That tension is real. We record it rather than resolving it.

5 The system

Table 1 pairs each model object with an identifier in the source. Every system claim below comes from source inspection at commit 1c3ad24. Two schema facts in Sections 6.8 and 6.9 discharge the hypotheses of Theorems 6.3 and 6.7.

AgentHero is a control plane for installed applications. Its repository documentation states the product boundary directly: the platform owns the Rust control plane, application discovery, executor contracts, runtime state, worker scheduling and process adapter dispatch. Product behaviour lives behind per-application manifests (CLAUDE.md, “Product Boundary”, commit 1c3ad24). That is a description of a harness in the sense of Section 2.1: it is what is left when the applications and the models are removed.

5.1 Model-to-source mapping

5.2 The knowledge layer

The certificate layer is the hook system. A hook is an installed plugin subscribed to canonical lifecycle events, and it has exactly two phases.

```
/// Hook execution phase.
pub enum HookPhase {
    /// Deterministic admission policy evaluated before canonical mutation.
    PrePolicy,
    /// Side-effect-capable handler evaluated after a canonical event commits.
    PostCommit,
}
```

Listing 1: crates/orchestrator/src/hooks.rs lines 36–44, attributes omitted; commit 1c3ad24.

The doc comments name the two roles the model needs. A `PrePolicy` hook is a certificate whose check gates a mutation; a `PostCommit` hook is a certificate whose witness is produced after the fact and retained. The repository’s own prose says the same: “Pre-policy hooks are synchronous and fail closed before a mutation. Post-commit hooks are created transactionally from PostgreSQL events and use durable attempts, leases, fencing, idempotency, and retained receipts” (README.md, lines 39 to 43).

Model object	Code identifier	Path
Stage set $\text{St}(G)$	<code>DagManifest.nodes</code>	<code>crates/dag-runtime/src/lib.rs:658</code>
Harness wiring G	<code>DagManifest, DagEdge</code>	<code>crates/dag-runtime/src/lib.rs:658, 636</code>
Global name map ν	<code>AppManifestAction.dag_type</code>	<code>crates/orchestrator/src/dag_apps.rs:588</code>
Compatibility test	<code>validate_unique_dag_types</code>	<code>crates/orchestrator/src/dag_apps.rs:2442</code>
Certificate c , phase	<code>HookPhase, HookManifest</code>	<code>crates/orchestrator/src/hooks.rs:39</code>
Claim identifier τ_c	<code>H00K_RECEIPT_SCHEMA</code>	<code>crates/orchestrator/src/hooks.rs:34</code>
Parameters σ_c	<code>HookRequest</code> identity fields	<code>crates/orchestrator/src/hooks.rs:404</code>
Scope $\text{scope}(c)$	<code>event_pattern, app_id, dag_type</code>	<code>crates/orchestrator/src/hooks.rs:1051</code>
Witness w	<code>HookReceipt</code>	<code>crates/orchestrator/src/hooks.rs:430</code>
Replay check chk_c	<code>validate_receipt</code>	<code>crates/orchestrator/src/hooks.rs:693</code>
Replay identity	<code>hook_idempotency_key</code>	<code>crates/orchestrator/src/hooks.rs:250</code>
Admission Adm	<code>evaluate_pre_policy_hooks</code>	<code>crates/orchestrator/src/hooks.rs:1042</code>
Per-stage Know_n	<code>DagToolPolicy</code>	<code>crates/dag-runtime/src/lib.rs:566</code>
Deployment map Φ , models	<code>provider_by_name</code>	<code>crates/llm-adapter/src/lib.rs:431</code>
Deployment map Φ , processes	<code>AppAdapter::Process</code>	<code>crates/orchestrator/src/dag_apps.rs:544</code>
Dispatch boundary	<code>AppAdapterRequest, AppAdapterResponse</code>	<code>crates/agent-runtime/src/app_protocol.rs:212, 370</code>
Components Comp_X , labelled by Φ	stage set of the manifest, labelled by <code>provider_by_name</code> or by <code>AppAdapter::Process</code>	<code>crates/llm-adapter/src/lib.rs:431; crates/orchestrator/src/dag_apps.rs:544</code>
Receipts Rcpt_X	<code>DurabilityBarrierReceipt</code>	<code>crates/dag-executor/src/lib.rs:141</code>
Receipt identities Rcpt_X	<code>DurabilityBarrierReceipt</code> fields <code>no_de_id, attempt</code>	<code>crates/dag-executor/src/lib.rs:141</code>
Sequencing $\sim_X$, total	<code>dag_events.idbigserial</code> , one sequence	<code>supabase/migrations/20260522000002_app_runtime_tables.sql:133</code>
Sequencing $\approx_X$, per row	<code>hook_deliveries.fence_token</code> , per-row counter	<code>supabase/migrations/20260824000002_plugin_hooks.sql:48</code>
Φ -reading check	<code>resolveManagedArtifactPath</code>	<code>packages/harness-runtime/src/workspace.mjs:22</code>

Table 1: Model objects and the identifiers realizing them in AgentHero at commit 1c3ad24. Section A records the line ranges and what each shows.

A subscription also carries a failure policy, which is the certificate’s declared behaviour when its check does not pass.

```
pub enum HookFailurePolicy {
    /// Deny the synchronous operation.
    Deny,
    /// Retry post-commit delivery with the persisted attempt policy.
    Retry,
    /// Retain the failed receipt without blocking canonical runtime progress.
    Continue,
}
```

Listing 2: `crates/orchestrator/src/hooks.rs` lines 57–66, attributes omitted; commit 1c3ad24.

The witness is a receipt, and it is closed: the type is declared with `serde(deny_unknown_fields)`, so a plugin cannot smuggle additional structure past the boundary.

```
/// Closed receipt returned by a hook plugin.
#[derive(Debug, Clone, Deserialize, Serialize)]
#[serde(deny_unknown_fields)]
pub struct HookReceipt {
    pub schema: String,          // stable receipt schema
    pub status: String,          // allow, deny, or completed
    pub plugin: String,
    pub hook_key: String,
    pub event_id: Option<i64>,    // absent for pre-policy admission
    pub idempotency_key: String,  // replay-safe identity
    pub output: Value,           // plugin-defined, closed to the platform
}
```

Listing 3: `crates/orchestrator/src/hooks.rs` lines 427–445, field comments condensed onto their fields; commit 1c3ad24.

Comparing Listing 3 with Theorem 3.3 gives the reading used throughout: τ_c is `schema`, and σ_c is the tuple `(plugin, hook_key, event_id, idempotency_key)` carried on the corresponding request. The scope `scope(c)` is the subscription’s event pattern together with its optional application and manifest filters, and the witness is the receipt itself. The check is a single function.

```
fn validate_receipt(request: &HookRequest, receipt: &HookReceipt,
    expected_status: &str) -> anyhow::Result<()> {
    ensure!(receipt.schema == HOOK_RECEIPT_SCHEMA, "hook receipt schema mismatch");
    ensure!(receipt.status == expected_status, "hook receipt status must be ...");
    ensure!(receipt.plugin == request.plugin, "hook receipt plugin mismatch");
    ensure!(receipt.hook_key == request.hook_key, "hook receipt key mismatch");
    ensure!(receipt.event_id == request.event.id, "hook receipt event mismatch");
    ensure!(receipt.idempotency_key == request.idempotency_key, "... mismatch");
    ensure!(receipt.output.is_object(), "hook receipt output must be an object");
    Ok(())
}
```

Listing 4: `crates/orchestrator/src/hooks.rs` lines 693–727, abridged; commit 1c3ad24.

Listing 4 is a receipt-shaped check in the sense of Theorem 3.6: six equalities between witness fields and request parameters, plus one syntactic test. It is total and decidable. This is the single most important identification in the paper, and also the one that bounds its claims. There is no theorem anywhere in Listing 3 or Listing 4 for a τ to denote.

The replay identity is a hash of transported data.

```

/// Deterministic replay identity for one subscription and canonical event.
pub fn hook_idempotency_key(hook_key: &str, event_id: i64) -> String {
    let mut digest = Sha256::new();
    digest.update(b"agenthero.hook.v1\0");
    digest.update(hook_key.as_bytes());
    digest.update([0]);
    digest.update(event_id.to_string().as_bytes());
    format!("{:x}", digest.finalize())
}

```

Listing 5: `crates/orchestrator/src/hooks.rs` lines 249–258, commit 1c3ad24.

A parallel function `pre_policy_idempotency_key` (line 1002) uses a distinct domain tag `agenthero.hook.pre-policy.v1` and a request identifier in place of the event identifier, so the two phases hash disjoint prefixes.

5.3 Admission

The admission predicate of Theorem 3.34 is implemented as a loop over matching subscriptions with two failure paths, both of which refuse.

```

match executor.invoke(&request).await {
    Ok(receipt) => {
        if let Err(error) = validate_receipt(&request, &receipt, "allow") {
            persist_policy_decision(pool, .., "deny",
                                   Some(&receipt), Some(..)).await?;
            bail!("pre-policy hook '{hook_key}' denied admission: {error}");
        }
        persist_policy_decision(pool, .., "allow", Some(&receipt), None).await?;
        receipts.push(receipt);
    }
    Err(error) => {
        persist_policy_decision(pool, .., "error", None, Some(..)).await?;
        bail!("pre-policy hook '{hook_key}' failed closed: {error}");
    }
}

```

Listing 6: `crates/orchestrator/src/hooks.rs` lines 1101–1142, abridged; commit 1c3ad24.

Two properties of Listing 6 matter. A receipt that fails `validate_receipt` is treated as a denial rather than as an error to be retried, and an executor that fails to produce a receipt at all is likewise a refusal. The decision, whichever way it goes, is persisted before the function returns, under a uniqueness constraint on (request, subscription). The selection query that produces the subscription list (lines 1051 to 1066) also selects `plugin_bundle_sha256`. Lines 1073 to 1077 raise an error when it is absent, so an unpinning plugin cannot be consulted for admission.

5.4 Fenced delivery

Post-commit certificates are delivered from a durable queue with leases and fences. A claim increments the fence, takes the lease, and is committed before the plugin runs.

```

let attempt      = row.try_get::<i32, _>("attempts")? + 1;
let fence_token = row.try_get::<i64, _>("fence_token")? + 1;
sqlx::query(
    "update hook_deliveries set state = 'leased', attempts = $2, fence_token = $3,

```

```

        lease_owner = $4, leased_until = now() + make_interval(secs => $5),
        updated_at = now()
    where id = $1",
).bind(id).bind(attempt).bind(fence_token).bind(lease_owner).bind(lease_seconds)
  .execute(&mut *tx).await?;
tx.commit().await?;

```

Listing 7: `crates/orchestrator/src/hooks.rs` lines 593–607, abridged; commit 1c3ad24.

Completion is conditional on the lease and fence still matching, and a mismatch is detected rather than ignored.

```

pub async fn complete_hook_delivery(pool: &PgPool, claim: &HookDeliveryClaim,
                                   receipt: &HookReceipt) -> anyhow::Result<()> {
    validate_receipt(&claim.request, receipt, "completed"?);
    let changed = sqlx::query(
        "update hook_deliveries set state = 'completed', receipt = $4, ...
        where id = $1 and state = 'leased'
        and lease_owner = $2 and fence_token = $3",
    ).bind(claim.id).bind(claim.lease_owner).bind(claim.fence_token)
    .bind(serde_json::to_value(receipt)?).execute(pool).await?.rows_affected();
    ensure!(changed == 1, "stale hook completion was fenced");
    Ok(())
}

```

Listing 8: `crates/orchestrator/src/hooks.rs` lines 642–662, abridged; commit 1c3ad24.

The parallel function `fail_hook_delivery` (line 665) applies the same fenced predicate, chooses between the `failed` and `dead_letter` states according to `HookFailurePolicy` and the attempt count, and computes a quadratic backoff capped at sixty seconds.

The other fence in the system is the durability barrier, which sits between a node’s admission and its handler. Its module documentation states the contract and its receipt carries the fence.

```

//! Canonical, lease-fenced commit-before-dispatch storage.

/// Commit one exact validated node-attempt identity before handler dispatch.
/// The signed worker lease is locked and checked in the same transaction as
/// the node projection and event insert. Returning a receipt therefore proves
/// PostgreSQL accepted the fence; an adapter event or in-memory observation is
/// never treated as acknowledgement.

receipt: DurabilityBarrierReceipt {
    schema: DURABILITY_RECEIPT_SCHEMA.to_string(),
    durable_ref: format!("postgres:dag_events/{event_id}"),
    dag_type: request.dag_type.clone(),
    manifest_hash: request.manifest_hash.clone(),
    node_id: request.node_id.clone(), attempt: request.attempt,
    fence: u64::try_from(event_id)?,
}

```

Listing 9: `crates/orchestrator/src/durability.rs` lines 1, 17–22 and 271–285, abridged; commit 1c3ad24.

Remark 5.1 (On a word in the quoted source). The module documentation reproduced in Listing 9 says that returning a receipt “proves PostgreSQL accepted the fence”. That sentence is the source’s and it is quoted because it is evidence of the design’s intent, not adopted as a claim of this paper. A successful commit is an operational acknowledgement from one store under its own failure

assumptions, and a paper cannot upgrade it to a deduction. Read the sentence as recording that the implementation treats a returned receipt, rather than an adapter event or an in-memory observation, as the acknowledgement it acts on. Nothing in Sections 3 and 6 depends on any stronger reading, and Law 5 is stated about what the store admits, not about what is true of the world.

The last line of Listing 9 is the whole of Theorem 6.3 in one expression. The fence of a durability-barrier receipt is the identifier of the row the barrier inserted into `dag_events`, and that column is declared `idbigserialprimarykey` (`supabase/migrations/20260522000002_app_runtime_tables.sql`, line 133). At this commit that path is a symbolic link to `agenthero/migrations/20260522000002_app_runtime_tables.sql`, whose lines are the ones cited. One sequence, one table, every installed application.

5.5 The deployment map

Φ is realized in two independent halves. For model-backed stages there is a name-keyed factory.

```
/// Build a provider by short name.
pub fn provider_by_name(name: &str, cfg: &ProviderConfig)
  -> Result<Arc<dyn LLMPProvider>, LLMEError> {
  match name {
    /* four feature-gated provider names */
    other => Err(LLMEError::Provider(format!("unknown provider: {other}"))),
  }
}
```

Listing 10: `crates/llm-adapter/src/lib.rs` lines 430–446, abridged; commit 1c3ad24.

For process-backed stages there is a manifest-declared adapter.

```
/// Adapter declaration for an installed DAGOps app.
#[serde(tag = "kind", rename_all = "snake_case")]
pub enum AppAdapter {
  /// Spawn a local process and exchange JSON over stdin/stdout.
  Process { command: String, runtime_commands: Vec<String>,
            host_commands: Vec<String>, environment: Vec<String>,
            args: Vec<String>, /* ... */ },
}
```

Listing 11: `crates/orchestrator/src/dag_apps.rs` lines 541–575, abridged; commit 1c3ad24.

Both halves are keyed by a name rather than by a stage. The assignment of a stage to a name lives in the application manifest and in configuration, so the code realizes the second factor of

$$\Phi: \text{St}(G) \xrightarrow{\text{manifest and configuration}} \text{Names} \xrightarrow{\text{resolution}} R.$$

This factorization is what makes Φ a parameter in Banu’s sense: the resolution half is fixed code, the assignment half is data that a deployment can change without touching (G, Know) . It is also why Law 6 is not automatic. A check that reads a root directory chosen by the adapter’s environment reads the left factor.

5.6 Registry composition

Installed applications are discovered from a directory of manifests, and the discovery function performs exactly the compatibility test of Theorem 3.21.

```

pub fn load_app_manifests_from_root(root: &Path)
  -> anyhow::Result<Vec<AppManifest>> {
  /* one manifest per subdirectory containing app.yaml */
  manifests.sort_by(|a, b| a.slug.cmp(&b.slug));
  validate_unique_dag_types(&manifests)?;
  Ok(manifests)
}

fn validate_unique_dag_types(manifests: &[AppManifest]) -> anyhow::Result<()> {
  let mut owners = BTreeMap::<&str, &str>::new();
  for manifest in manifests { for action in &manifest.actions {
    if let Some(owner) = owners.insert(action.dag_type.as_str(),
                                       manifest.slug.as_str()) {
      if owner != manifest.slug {
        bail!("dag_type '{}' is declared by both app
              '{owner}' and app '{}'", ..);
      } } } }
  Ok(())
}

```

Listing 12: crates/orchestrator/src/dag_apps.rs lines 1410–1427 and 2442–2458, abridged; commit 1c3ad24.

The set N of global names of Theorem 3.20 is the set of manifest type identifiers, ν is the assignment of a type identifier to each declared action, and the compatibility test is the injectivity check in Listing 12. The check is performed over all installed applications at once, before any of them is admitted, which is the implementation of a partial monoidal product refusing an inadmissible pair.

5.7 The harness runtime boundary

Two functions in the JavaScript harness runtime carry structure the model needs. The first normalizes raw provider output before it reaches anything else. `sanitizeProviderSessionEvents` (packages/harness-runtime/src/provider-session.mjs, line 8) is documented as converting provider JSONL into “bounded, content-safe operator session events”, against explicit budgets declared at lines 1 to 3 (`MAX_VISIBLE_TEXT_CHARACTERS`, `MAX_VISIBLE_RECORD_CHARACTERS`, `STREAM_FLUSH_BYTES`). It is wrapped by `ProviderSessionCapture` (line 114), which suppresses repeated tool call identifiers. This sits between the model and the rest of the architecture and belongs to neither G nor Know: it is part of the realization.

The second is a check, and it is the counterexample of Theorem 3.32(2) in the field.

```

export function resolveManagedArtifactPath(artifact, workspaceRoot) {
  const expected = artifact.metadata?.content_sha256 ?? artifact.metadata?.sha256;
  const artifactStore = process.env.AGENTHERO_ARTIFACT_STORE_DIR;
  const allowedRoots = [workspaceRoot, artifactStore]
    .filter(Boolean).map(r => resolve(r));
  /* ... URI parsing ... */
  if (!allowedRoots.some((root) => isStrictDescendant(root, path)))
    throw new Error('artifact path is outside an authorized root: ${path}');
  if (!existsSync(path) || !lstatSync(path).isFile()
      || lstatSync(path).isSymbolicLink())
    throw new Error('artifact path is missing, non-regular, or symbolic: ${path}');
  if (expectedHash) {
    const actual = createHash("sha256").update(readFileSync(path)).digest("hex");
    if (actual !== expectedHash)

```

```

        throw new Error('artifact content hash mismatch ...');
    }
    return path;
}

```

Listing 13: `packages/harness-runtime/src/workspace.mjs` lines 22–30 and 51–65, abridged; commit 1c3ad24.

Listing 13 performs three tests. The content hash comparison factors through (G, Know) : the expected hash is carried in the artifact reference, which is transported data. The containment and symbolic link tests do not: `allowedRoots` is built from the workspace root and from an environment variable naming the artifact store, both of which are deployment data.

The corresponding certificate $c_{\text{art}} \in \text{Know}$ has claim identifier `managed-artifact`, parameters for the artifact URI and expected digest, and scope equal to the consuming stage. Its witness records the resolved path, actual digest, file kind, and symlink status. The checker requires a regular non-symlink file, digest equality when a digest is supplied, and strict containment under a root in the consuming stage’s deployment record $\Phi(n)$. The first two clauses are deployment-independent; containment reads Φ .

6 Law checking

The seven laws are tested against the source identifiers in Table 1. Each result is an implementation claim from source inspection at commit 1c3ad24, not an observation of a running system. The durability and hook subsections then supply the schema hypotheses required by Theorems 6.3 and 6.7.

6.1 Law 1: separability

Hook certificates are rows in a `hook_subscriptions` table, selected at admission time by a query on `phase`, `enabled` and the event pattern (`crates/orchestrator/src/hooks.rs`, lines 1051 to 1066). They are installed by a separate manifest schema `agenthero.hook.v1` (line 30), whose receipt counterpart `HOOK_RECEIPT_SCHEMA` is declared at line 34. Nothing about installing, listing or removing a hook requires editing a wiring: the certificate layer is addressable on its own. Compared with the situation Part III reports for CatDB, where policy is evaluated inline with planning and provenance is attached to individual field values inside an executed plan, this is a genuine separation.

`DagToolPolicy` (`crates/dag-runtime/src/lib.rs:566`) carries budget, approval, network, filesystem, subprocess, and credential policy inside the manifest. Stage scope alone does not make that record a certificate under Theorem 3.3: no claim identifier, witness, or replay checker is specified here. We therefore exclude it from `Know` rather than using it as a counterexample to separability. Law 1 holds for the identified hook-certificate layer; whether tool policy admits a complete certificate interpretation remains open.

6.2 Law 2: receipt identity replay

`validate_receipt` (Listing 4) is total, decidable, and compares six identity fields plus one structural test. It is called on both paths: at admission with expected status `allow` (line 1103) and at completion with expected status `completed` (line 647). The receipt type is closed under `deny_unknown_fields`, so the check cannot be evaded by adding fields. A mismatch produces a refusal, not a warning.

6.3 Law 3: deterministic replay identity

`hook_idempotency_key` (Listing 5) is a SHA-256 digest over a domain tag, the subscription key and the canonical event identifier, with explicit zero-byte separators between fields, so distinct field tuples cannot produce the same input string. `pre_policy_idempotency_key` uses a different domain tag. The store enforces the same identity independently: `hook_deliveries` carries `unique(subscription_id,event_id)` (`supabase/migrations/20260824000002_plugin_hooks.sql`, line 57), so a second delivery row for the same pair cannot be created.

One qualification is required by Theorem 3.30. The identity is a function of the subscription key and an event identifier assigned by the store at insert time. Under an architecture morphism the subscription key is transported, but the event identifier is not architecture data. The certificate is therefore replay-stable relative to a fixed event stream, and not across two runs of the same architecture. This is a real narrowing: Law 3 is only partially satisfied.

6.4 Law 4: fail-closed admission

Listing 6 refuses on three distinct conditions: an executor error, a receipt that fails validation, and, before either, a subscription without a pinned plugin bundle hash. Each refusal propagates out of `evaluate_pre_policy_hooks` and therefore out of the mutation that called it, and there is no branch on which a failed check yields admission. The three differ in what they leave behind. The executor-error and invalid-receipt paths call `persist_policy_decision` before raising, so a decision row survives. The missing-bundle path raises at lines 1073 to 1077, before the request is built, so nothing is persisted for it.

By Theorem 3.34 this has a consequence the implementation cannot avoid. Installing an additional enabled `pre_policy` subscription whose pattern matches an event is a morphism that preserves every existing certificate and can turn an admitted mutation into a refused one. The selection query at line 1053 returns the new row, and the loop conjoins its decision. A harness that satisfies Law 4 is one whose admission predicate is antitone in `Know`, and no amount of certificate preservation repairs that.

6.5 Law 5: fence exclusion

For writes to the canonical store the law holds. `complete_hook_delivery` (Listing 8) and `fail_hook_delivery` both predicate their update on `id, state='leased', lease_owner` and `fence_token`. Both assert `rows_affected()==1`, with the messages “stale hook completion was fenced” and “stale hook failure was fenced”. A superseded writer’s update matches no row and the mismatch is raised. The claim path (Listing 7) takes the row with `forupdate...skiplocked` inside a transaction and increments the fence before committing, so two dispatchers cannot hold the same fence.

The law concerns whether a stale writer can update the canonical record, and that property holds. External effects form a separate limitation. A plugin invoked under a claim runs as a child process. `InstalledPluginHookExecutor` is declared at line 749 of the same file, and its `invoke` method at line 1149 spawns the resolved process with `env_clear()` at line 1163 and an explicit environment whitelist. That is environment isolation and not filesystem or network isolation, neither of which appears at this boundary. The plugin may perform an external effect before its completion is fenced out. The platform offers a deduplication key for this case, the `idempotency_key` field on `AppAdapterRequest` (`crates/agent-runtime/src/app_protocol.rs:233`). Its own documentation places the obligation on the adapter: it is a “stable key adapters use to deduplicate retried external side effects”. The field is data the platform supplies, not a check the platform performs, so whether a retried effect is suppressed is decided outside the harness. Fencing protects the canonical record,

not the world.

6.6 Law 6: Φ -independence

Listing 13 constructs its authorized root set from `workspaceRoot` and from `process.env.AGENTHERO_ARTIFACT_STORE_DIR`, and refuses any path that is not a strict descendant of one of them. Both are deployment data in the sense of Section 5.5: they are determined by where the harness placed the application and which store the environment names, not by the wiring or the certificate set. The check therefore reads Φ . This particular certificate is not replay-stable: redeploying the same (G, Know) against a different artifact root changes whether a path lying under one root but not the other passes. Theorem 3.32(2) gives the same shape of counterexample but is not needed as a universal inference.

The same function contains a check on the other side of the line. The content hash comparison against `artifact.metadata.content_sha256` depends only on transported data, and by Theorem 3.32(1) it is replay-stable. One function, two certificates, one preserved and one not. This is the sharpest available statement of what Law 6 costs and why a realization should separate the two tests rather than conjoining them.

6.7 Law 7: name disjointness

`validate_unique_dag_types` (Listing 12) is called by `load_app_manifests_from_root` at line 1425, before the manifest list is returned, and raises when two distinct application slugs declare the same type identifier. The enumeration entry points reach it: `load_app_manifests` (line 1405) delegates to it directly, and `registered_dag_apps` (line 2461) calls `load_app_manifests`. This is Theorem 3.21’s decision procedure, realized.

One qualification belongs here. The check is a property of enumeration, not of loading. `load_app_manifest_by_slug_from_root` (line 1443) reads a single manifest and never sees the others, so it cannot detect a collision and does not try to. The implementation decides admissibility of the composite whenever it forms the composite, which is what Theorem 3.21 requires. It says nothing when one factor is loaded alone, which the proposition does not require.

6.8 Laxness on the durability fragment

The fragment result follows. Everything in it is determined by the stage sets and by one schema fact, and no part of it refers to a run. The schema fact enters only as a named hypothesis of Theorem 6.3, that the composite’s sequencing relation is total, and it is discharged for **AgentHero** in the paragraph following the proof. The theorem does not depend on how, or whether, it is discharged. Theorem 6.7 is built the same way around the opposite hypothesis, that delivery fences are per row.

Definition 6.1 (The durability interpretation). Let $\mathbf{Arch}_{\text{db}} \subseteq \mathbf{Arch}$ be a monoidal subcategory of architectures realised by the global-sequence durability store described below, with durability-barrier certificates whose witnesses are values of `DurabilityBarrierReceipt`; it is closed under $\otimes$ and contains I . Its morphisms are the architecture morphisms between those realisations. Define

$$A_{\text{db}}(\mathcal{A}) = (\text{St}(G_{\mathcal{A}}), \Phi_{\mathcal{A}}, \text{St}(G_{\mathcal{A}}) \times \mathbb{N}^+, \text{fld}_{\text{db}}, \nabla_{\mathcal{A}}),$$

where the component set is the stage set, each component labelled by its realizer $\Phi_{\mathcal{A}}(n)$. A receipt identity is a pair (stage, attempt number), carrying the barrier schema string and the identity fields `node_id` and `attempt`, so $\text{fld}_{\text{db}}(n, i) = (\text{barrier-schema}, n, i)$. The relation $\nabla_{\mathcal{A}}$ is the total equivalence relation on receipt identities. On a morphism f , set $A_{\text{db}}(f) = (f_{\text{St}}, f_{\text{St}} \times \text{id}_{\mathbb{N}^+})$.

The receipt identity set is the set of shapes a barrier receipt may take, determined by $G_{\mathcal{A}}$ alone. It is countably infinite, and it is not a set of receipts that were produced. The relation $\nabla_{\mathcal{A}}$ is total because the barrier's fence is drawn from one database sequence, which is a fact about the schema, recorded after Theorem 6.3. Component labels are not constrained by **Sys** morphisms, mirroring Theorem 3.11: an interpretation records Φ as a labelling that a translation may change.

Proposition 6.2. A_{db} is a functor $\mathbf{Arch}_{\text{db}} \rightarrow \mathbf{Sys}$.

Proof. The component map f_{St} is injective because f is a morphism of **Arch**, which is what **Sys** requires of it. The receipt map is not required to be injective and is in fact injective here. The schema string is constant and the identity fields are carried along f_{St} by construction. Labels are unconstrained. The sequencing implication holds vacuously in the strong sense that the target relation is total. Identities go to identities, and $A_{\text{db}}(gf) = ((gf)_{\text{St}}, (gf)_{\text{St}} \times \text{id}) = A_{\text{db}}(g) \circ A_{\text{db}}(f)$ since $(gf)_{\text{St}} = g_{\text{St}}f_{\text{St}}$. $\square$

Theorem 6.3 (The durability fragment is lax and not strong). *Let $\mathcal{A}, \mathcal{B} \in \mathbf{Arch}_{\text{db}}$. By the definition of this subcategory, the barrier co-sequences every receipt identity of each architecture and of their composite. Then:*

1. the canonical comparison $\mu_{\mathcal{A}, \mathcal{B}}: A_{\text{db}}(\mathcal{A}) \boxtimes A_{\text{db}}(\mathcal{B}) \rightarrow A_{\text{db}}(\mathcal{A} \otimes \mathcal{B})$ is a morphism of **Sys**, natural in both arguments, and its two underlying functions are bijections;
2. if $\text{St}(G_{\mathcal{A}})$ and $\text{St}(G_{\mathcal{B}})$ are both nonempty, then $\mu_{\mathcal{A}, \mathcal{B}}$ is not an isomorphism of **Sys**.

Together with $\eta = \text{id}_J$, this makes A_{db} lax monoidal and not strong.

Proof. (1) On components, μ_{Comp} is the identity of $\text{St}(G_{\mathcal{A}}) \sqcup \text{St}(G_{\mathcal{B}}) = \text{St}(G_{\mathcal{A} \otimes \mathcal{B}})$, by Theorem 3.17. On receipt identities, μ_{Rcpt} is the canonical distributivity bijection

$$(\text{St}(G_{\mathcal{A}}) \times \mathbb{N}^+) \sqcup (\text{St}(G_{\mathcal{B}}) \times \mathbb{N}^+) \xrightarrow{\cong} (\text{St}(G_{\mathcal{A}}) \sqcup \text{St}(G_{\mathcal{B}})) \times \mathbb{N}^+,$$

which preserves the schema string and both identity fields because it changes neither the stage nor the attempt number. The sequencing implication holds because the target relation is total. Naturality in each argument holds because μ_{Comp} is an identity and μ_{Rcpt} is the canonical distributivity map, which is natural in each factor. The unit η is the identity of J because $A_{\text{db}}(I)$ has five empty components and equals J . The associativity and unit coherence diagrams commute because every arrow in them is a canonical coproduct or distributivity comparison and those satisfy the corresponding coherence in **Set**.

(2) Suppose $\mu = \mu_{\mathcal{A}, \mathcal{B}}$ had an inverse μ^{-1} in **Sys**. Since μ_{Rcpt} is a bijection, μ_{Rcpt}^{-1} is its set-theoretic inverse, and being a morphism of **Sys** it must satisfy the sequencing implication. Choose stages $n_{\mathcal{A}} \in \text{St}(G_{\mathcal{A}})$ and $n_{\mathcal{B}} \in \text{St}(G_{\mathcal{B}})$, which exist by hypothesis, and put $r = (n_{\mathcal{A}}, 1)$ and $s = (n_{\mathcal{B}}, 1)$ in $\text{St}(G_{\mathcal{A} \otimes \mathcal{B}}) \times \mathbb{N}^+$. Then $r \sim s$ in the target, because that relation is total. So $\mu_{\text{Rcpt}}^{-1}(r)$ and $\mu_{\text{Rcpt}}^{-1}(s)$ must be related in the source. But $\mu_{\text{Rcpt}}^{-1}(r)$ lies in the left summand and $\mu_{\text{Rcpt}}^{-1}(s)$ in the right, and the source relation $\nabla_{\mathcal{A}} \sqcup \nabla_{\mathcal{B}}$ relates nothing across summands. This is a contradiction, so no inverse exists. $\square$

Remark 6.4. The proof shows something slightly sharper than laxness. The comparison μ is a bijection on both underlying sets and still fails to be invertible in **Sys**, in the way a continuous bijection can fail to be a homeomorphism. Composing two applications loses nothing, but it co-sequences receipts that previously belonged to separate sequences. A morphism cannot remove this added structure.

The hypothesis that the composite’s sequencing relation is total is the implementation-level claim, and it is a claim about a schema. It holds for **AgentHero** at commit 1c3ad24 because the barrier’s fence is `u64::try_from(event_id)?` (Listing 9), the event identifier is the primary key of `dag_events`, and that column is declared `idbigserialprimarykey` with no application scoping in its key (`supabase/migrations/20260522000002_app_runtime_tables.sql`, line 133). A `bigserial` column draws from one database sequence, so every row of the table, whichever application produced it, is co-sequenced with every other. A later migration states the intent in prose: “`dag_events.id` is allocated by PostgreSQL’s non-transactional sequence before this trigger runs. Reusing that value for an omitted `run_seq` preserves a strict order within every app run (gaps are allowed)” (`supabase/migrations/20260827000003_lock_free_dag_event_sequence.sql`, lines 3–5). The gaps within one application’s fences are the values other applications took, which is the same fact seen from the other side. Because PostgreSQL allocates sequence values before commit, the field establishes numeric allocation order. It does not by itself establish commit order or the order in which a replay consumer processes rows.

The two results just proved are instances of one criterion, which is where the categorical content sits. Say that an interpretation A has *local sequencing* on a pair $\mathcal{A}, \mathcal{B}$ when the composite’s sequencing relation is exactly the transport, along $\mu_{\mathcal{A}, \mathcal{B}}$, of the disjoint-sum relation on $A(\mathcal{A}) \boxtimes A(\mathcal{B})$. Composition then adds neither cross-summand pairs nor new pairs within a summand.

Theorem 6.5 (Strength is local sequencing). *Let $A: \mathbf{Arch} \rightarrow \mathbf{Sys}$ be an interpretation whose comparison $\mu_{\mathcal{A}, \mathcal{B}}$ is bijective on components and on receipt identities for every pair. Then $\mu_{\mathcal{A}, \mathcal{B}}$ is an isomorphism of $\mathbf{Sys}$ if and only if A has local sequencing on $\mathcal{A}, \mathcal{B}$. Consequently A is strong on a full monoidal subcategory exactly when it has local sequencing on every pair in it.*

Proof. Write $\mu = \mu_{\mathcal{A}, \mathcal{B}}$ and let μ_{Rcpt} be its bijection on receipt identities. Since μ is a morphism, μ_{Rcpt} preserves the sequencing relation. μ is an isomorphism if and only if μ_{Rcpt}^{-1} is also relation-preserving, that is, if and only if μ_{Rcpt} reflects the relation. Reflection fails exactly when some source pair is unrelated while its images are co-sequenced in the composite. Together with forward preservation, absence of such a pair is exact agreement with the transported disjoint-sum relation, which is local sequencing. $\square$

Theorem 6.3 and Theorem 6.7 are the two sides of Theorem 6.5, and the criterion is what makes them more than a pair of observations about integer counters. Strength of an interpretation is not a property of the architectures being composed. It is a property of the sequencing resource the realization gives them, and it holds precisely when that resource is partitioned along the same boundary the architectures are. A design that wants strength must therefore make its ordering resource local to a component; a design that wants a single cross-application allocation order must give strength up. Treating that order as replay order additionally requires consumers to sort by it and accept gaps and commit-order inversions. No third option exists for the modeled relation, because the criterion is an equivalence.

Remark 6.6. The content of Theorem 6.3 is operational, not decorative. Two applications installed into one **AgentHero** deployment share a sequencing resource. In the composite, any barrier receipt of one application is ordered against any barrier receipt of the other; in the pair, no such ordering exists. That relation is created by composition and belongs to neither factor, which is what a non-invertible comparison records. A harness that wanted a strong interpretation would have to give each application its own sequence, and would then lose the cross-application allocation order the store currently provides.

It is fair to observe that, read at the level of this one system, the theorem reduces to the remark that rows sharing a `bigserial` column interleave. The reduction is the point: the model was chosen

so that the question “is this interpretation strong” reduces to a question a database administrator can answer by reading a schema. What Theorem 6.5 adds is that the reduction is exact and general. Strength is not merely implied by locality of the sequencing resource, it is equivalent to it, for every interpretation whose comparison is bijective. So a designer does not have to reason about monoidal functors to decide the question, and a theorist does not have to inspect a schema to know what deciding it settles.

6.9 Strength of the hook-fence abstraction

The same harness is strong elsewhere, and the contrast is instructive.

Proposition 6.7. *Fix a set E of canonical event identifiers. Let $\mathbf{Arch}_{\text{hk}} \subseteq \mathbf{Arch}$ be the full monoidal subcategory of architectures whose certificates are post-commit hook certificates, closed under $\otimes$ and containing I , and define*

$$\begin{aligned} A_{\text{hk}}(\mathcal{A}) &= (\text{St}(G_{\mathcal{A}}), \Phi_{\mathcal{A}}, \text{Know}_{\mathcal{A}} \times E \times \mathbb{N}^+, \text{fld}_{\text{hk}}, \approx_{\mathcal{A}}), \\ \text{fld}_{\text{hk}}(c, e, i) &= (\tau_c, e, i), \\ (c, e, i) \approx_{\mathcal{A}} (c', e', i') &\iff c = c' \text{ and } e = e', \end{aligned}$$

with $A_{\text{hk}}(f) = (f_{\text{St}}, f_{\text{Know}} \times \text{id}_E \times \text{id})$. Then A_{hk} is a functor and $\mu_{\mathcal{A}, \mathcal{B}}^{\text{hk}}$ is an isomorphism for all $\mathcal{A}, \mathcal{B} \in \mathbf{Arch}_{\text{hk}}$, so A_{hk} is strong.

Proof. Functoriality is as in Theorem 6.2. If two source receipts have equal certificate and event coordinates, their images under the function $f_{\text{Know}} \times \text{id}_E \times \text{id}$ have equal coordinates as well; injectivity of f_{Know} is unnecessary. Condition (M1) preserves fld_{hk} . For the comparison, $\text{Know}_{\mathcal{A} \otimes \mathcal{B}} = \text{Know}_{\mathcal{A}} \sqcup \text{Know}_{\mathcal{B}}$ by Theorem 3.17, so $\mu_{\text{Rcpt}}^{\text{hk}}$ is the canonical bijection $(\text{Know}_{\mathcal{A}} \sqcup \text{Know}_{\mathcal{B}}) \times E \times \mathbb{N}^+ \cong (\text{Know}_{\mathcal{A}} \times E \times \mathbb{N}^+) \sqcup (\text{Know}_{\mathcal{B}} \times E \times \mathbb{N}^+)$ read in the other direction. Two receipt identities of the composite are related exactly when they agree in their first two coordinates, and agreeing in the first coordinate forces them into the same summand, since the two certificate sets are disjoint. Hence the composite’s relation has no cross-summand pairs and coincides with the disjoint sum relation under the bijection. So both μ^{hk} and its set-theoretic inverse preserve the relation, and μ^{hk} is an isomorphism. $\square$

The certificate coordinate c retains its scope, but A_{hk} does not use that scope to restrict the event coordinate e . The proposition therefore concerns only the partition of possible receipt identities by fence resource. It does not prove that scope-based event eligibility is preserved by composition, nor does it model the set of deliveries that actually occur. A stronger operational target would need an architecture-indexed event set and an explicit eligibility relation between certificate scopes and event targets.

The implementation-level claim behind Theorem 6.7 is that `hook_deliveries.fence_token` is a per-row counter rather than a shared sequence: the column is declared `bigintnotnulldefault0` ([supabase/migrations/20260824000002_plugin_hooks.sql](#), line 48), the same file constrains a row to one subscription and one event with `unique(subscription_id,event_id)` (line 57), and Listing 7 sets the token to that row’s previous value plus one. Two delivery rows are therefore never co-sequenced, which is the schema fact the proposition needs.

Law	Status	Basis at commit 1c3ad24
Law 1	satisfied on the identified certificate set	Hook subscriptions are separately addressable rows; <code>DagToolPolicy</code> is not classified as a certificate
Law 2	satisfied	<code>validate_receipt</code> , six identity equalities plus one structural test, called on both paths
Law 3	partial	<code>hook_idempotency_key</code> is deterministic in transported data plus a store-assigned event identifier
Law 4	satisfied	<code>evaluate_pre_policy_hooks</code> refuses on executor error, invalid receipt and unpinned bundle
Law 5	satisfied	Stale store writes are fenced and the mismatch is raised; external effects are a separate limitation
Law 6	violated	<code>resolveManagedArtifactPath</code> reads the workspace root and the artifact store environment variable
Law 7	satisfied	<code>validate_unique_dag_types</code> runs inside the enumeration loaders

Table 2: Status of the seven laws against `AgentHero` at commit 1c3ad24.

7 Layer composition

7.1 Layer contribution

Parts I to III each describe a component of an architecture. None of them describes the act of installing that component somewhere, deciding whether an operation on it is admitted, or recording that the operation happened. Those three are what the harness adds, and they correspond exactly to Φ , to the admission predicate, and to the witness set.

The addition is not free. Each of the three brings a failure mode that the component layers do not have. Φ brings Theorem 3.32(2): once deployment is a parameter, checks that read it stop being stable. Admission brings Theorem 3.34: once refusal is conjunctive and fail-closed, adding a certificate changes behaviour even though every old certificate is preserved. Recording brings Theorem 6.3: once the record is ordered by a shared resource, composition creates information that neither factor had.

7.2 Part I dependency

Part I models a memory store by two coalgebras on the same carrier [10]: a store coalgebra for a deterministic input-output functor whose alphabet is the memory interface extended by an allocation oracle, and a lineage coalgebra whose final semantics is the ancestry of a record. This Part uses neither construction. It uses only the boundary the two of them fix. A certificate is something checked; a memory record is something remembered. As stated in Part I, the schema registry and structured-data validation belong on the Know side and the stored lineage belongs on the S side, and Part I’s separation theorem shows that `ContextFS`’s recorded state is single-axis, so a retroactive correction and a belated recording are indistinguishable in it. Five of Part I’s eight law clauses fail against its own system. That result sets the standard: a pillar paper in this series reports the failures rather than restricting attention to the fragment that works.

The dependency is architectural rather than realized. At commit 1c3ad24 the `AgentHero` workspace manifest (`Cargo.toml`, lines 1–72) lists eight member crates and its third-party dependency set, and neither names `ContextFS`; the harness runtime package (`packages/harness-runtime/package.json`, lines 1–21) declares no dependency block at all. A case-insensitive search over the Rust, TOML and JavaScript sources likewise returns no occurrence of the name. The harness does carry a store-side projection of session content, `agent_session_memory_chunks` (`supabase/`

migrations/20260824000001_agent_session_memory.sql, line 4), keyed by `source_event_id`, but that is store state and not a coalgebra in Part I’s sense; it has no evolve, merge, split or supersede operations and no lineage edges. The composition of the memory pillar with this harness is therefore a statement about the series’ model, not about a shipped integration, and we say so rather than implying otherwise.

7.3 Part II dependency

Part II’s operad is realized in the same repository as this Part’s harness, which makes the boundary between them a matter of discipline rather than of packaging. The division we adopt is the one recorded in both contracts: Part II owns `NodeHandler` and the reading of a validated manifest as an operadic composite; this Part owns `AppAdapterRequest`, `AppAdapterResponse` and `provider_by_name`, which are the deployment boundary rather than the composition law.

Three results from Part II are used here [11]. Validation establishes a finite acyclic dependency graph but not the rooted output and unique supplier map needed to determine one operadic operation. A plain operad splits a multi-output invocation into coordinate generators, while a PROP preserves it as one operation. Actual sibling-output disjointness is sufficient for order-independent merging, while a collision carrying distinct values is a counterexample; declared disjointness is insufficient unless returned value keys are confined. That statement is the operad-level analogue of Theorem 3.21: in both cases a composition law is total on the model and partial on the implementation, and in both cases the side condition concerns disjointness of names. The two conditions are not the same condition, since one is about actual outputs within a manifest and the other about type identifiers across applications, but both expose collisions in a shared namespace.

7.4 Part III dependency

The prospectus asks whether the series has one wiring object or two. The answer this Part adopts, following Part III [12], is two: $\mathcal{W}$ and $\mathcal{O}$ are distinct operads sharing a colour vocabulary, and there is no checkable identification of `CatDB`’s port-type alphabet with `AgentHero`’s string-keyed JSON ports at the pinned commits. Part III’s type-erasure result is in two halves, and both are needed here. In one direction, any assignment sending 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 as soon as two distinct field sets share a key. In the other direction, nothing checks an assignment at all, because the manifest edge type records a source and a target node identifier and no port datum.

This Part therefore does not fix one G for the series. G is a component slot in Theorem 3.8, and different architectures fill it with different wiring objects of the same shape. Theorem 3.1 names the harness-level filler and defines it as a general wiring in Part III’s operad $\mathcal{W}_T$, not as a tree wiring; nothing in Theorems 3.10, 3.18, 3.32 and 6.3 depends on which filler is chosen, because all four are proved from the stage set, the certificate set and the deployment map alone. That is a deliberate feature of the presentation: the results transfer to any Part’s wiring, which is what makes them series-level results rather than `AgentHero`-level ones.

Theorem 3.2 adds a second, independent difference between the two layers. The protocol system’s wirings are tree wirings and the harness’s are not, so the two Parts are studying different fragments of the same operad, and a result proved by induction over a tree in Part III has no automatic counterpart here. The series therefore does not describe one wiring object seen from four angles.

On the Know side the two Parts divide the evidence. Part III reports that `CatDB`’s policy decisions and field provenance records are certificate-shaped, in a sense this Part would make precise as Theorem 3.6, but are not separately addressable, so the G and Know components are

not distinguishable at the crate level. This Part reports (Section 6.1) that **AgentHero** separates its modeled hook certificates. Per-operation tool policy is not classified as a certificate here because a complete witness and checker mapping was not established.

7.5 Dependency graph

This Part imports Part I’s coalgebra as a component realized inside a harness, Part II’s operad as the composition law for stages within one application, and Part III’s wiring operad $\mathcal{W}_\top$ as the shape of G , on the non-tree fragment described in Theorem 3.2. It exports Theorem 3.3, Theorem 3.5, Theorem 3.8 and the reading of certificate preservation as replay stability to Parts II and III, both of which cite them without restating them. Nothing proved in this Part is used in the proof of any result in Parts I to III, so the import graph restricted to formal dependencies is acyclic, with this Part as its unique sink. The cycles the prospectus asks each Part to state are operational rather than formal: a Part II operation may call a Part I memory operation at runtime, and a Part III wiring may carry a Part II skill as a node, without either Part importing the other’s definitions.

8 Limits and counterexamples

8.1 Receipt-shaped certificates

Banu’s Definition 1 requires τ , a theorem statement. Listing 3 has **schema**, a version string. Listing 4 compares identities. The **output** field is documented as “plugin-defined closed-to-the-platform output”, which means the platform does not interpret it, so no logical content can be recovered from it either.

The consequence bounds the enterprise as applied here. Preservation of a certificate means, by Theorem 3.30, 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 harness has Know-shaped machinery with Know-content that Theorem 3.7 shows to be blind to everything except identity and record shape. What remains true is narrower and still worth stating: the machinery is the right shape, the checks are total and decidable, the identities are deterministic, and the results of Section 3 apply. Section 10 states what filling τ would require.

8.2 Sorting hypothesis

Theorem 3.32(1) needs the value set to be sorted, so that a stage identifier can never be confused with a payload string. The hook certificates satisfy it, because **HookRequest** and **HookReceipt** give names and payloads distinct typed record positions (Theorem 3.4). The manifest port vocabulary does not, and the failure is not hypothetical.

Example 8.1 (Unsorted values break transport). Let $\mathcal{A}$ have stages $\{n_1, n_2\}$ and a certificate c with a single parameter p , $\sigma_c(p) = n_2$, and $\text{chk}_c(\Phi, w) = \top$ if and only if the field $w\ell$ equals $\sigma_c(p)$. Suppose the value set is unsorted, so that the string n_2 may also occur as a payload. Let f rename n_1 to n_2 and n_2 to n_3 . Then f_* sends $\sigma_c(p) = n_2$ to n_3 , while a witness w whose field ℓ carried the payload string n_2 has $f_\#$ leaving it as n_2 , since $f_\#$ renames only name occurrences and cannot tell this one apart. So $\text{chk}_c^A(w) = \top$ and $\text{chk}_{f_{\text{Know}(c)}}^B(f_\#w) = \perp$: transport has destroyed the certificate. Under the sorting hypothesis the payload occurrence lives in V_{pl} , the equality could never have held, and the example does not arise.

As stated in Part II, **AgentHero**’s manifest ports are keyed by arbitrary strings with JSON payloads, so a certificate written against manifest ports rather than against hook records has no

protection from Theorem 8.1. This bounds the theorem’s reach inside the very system it is about, and the boundary runs along the same line as Part II’s: wherever the port vocabulary is untyped, the structural results weaken. The reach can be stated exactly. Every certificate identified in Sections 5.2 and 5.4 has a witness that is a closed record with typed positions, a hook receipt or a durability receipt, so the sorting hypothesis holds for all of them and Theorem 3.32(1) applies to all of them; the harness at this commit carries no certificate whose witness is a manifest port map. A certificate of that kind would have to establish the hypothesis first, by typing its name-valued fields, and the theorem says nothing about it until it does.

8.3 Redeployment counterexample

Section 6.6 exhibits `resolveManagedArtifactPath` as a check that reads the deployment map, and Theorem 3.32(2) shows such checks are not replay-stable. The model predicts this rather than being embarrassed by it. The containment test is security-relevant, so the certificates most likely to read Φ are the ones a deployment cares most about. Moving the deployment-relative part of such a test into the wiring, by making the authorized root a declared port type rather than an ambient environment variable, is what Part III reports `CatDB` does for authorized field sets, and is the available remedy.

8.4 Lax composition

By Theorem 6.5 the interpretation cannot be made strong on the durability fragment without changing the store design. A harness that draws its fences from one sequence violates the strong-functor requirement. The single shared order supplied by that design is also what obstructs strength.

8.5 Reference vocabulary

Banu’s four-pillar table names concrete components of his own reference implementation: `BiTemporalMemory` and `RunContext` for Memory, `SkillStage` and `PatternTemplate` for Skills, `WiringDiagram` for Protocols, and `SkillOrganism` for Harness. A recursive search for these six identifiers across the Rust, JavaScript and TypeScript sources of `AgentHero` at commit `1c3ad24` returns no match, and Parts I to III report the same for their own systems. The search is stated exactly in Section A, because an absence is not exhibited by any line. Nothing in this series implements that vocabulary. The claim is only that a system built without knowledge of it independently exhibits a structure the vocabulary names, which is a weaker and more interesting claim than implementation would be, and a reader should hold it to the weaker standard.

8.6 Fence scope

Section 6.5 records that a plugin can perform an external effect and then be fenced out of recording it. The model as stated has no vocabulary for this: **Sys** objects carry components and receipts, and an unrecorded effect is neither. Extending **Sys** with an effect monad and asking which of Law 5’s guarantees survive is the most concrete gap between this model and the systems it is about.

8.7 Missing system composition

The four-pillar reading describes a harness as the layer that composes memory, skills and protocols. At commit `1c3ad24` `AgentHero` composes one of the three. Its workspace manifest declares eight

member crates and a third-party dependency set naming neither ContextFS nor CatDB (`Cargo.toml`, lines 1–72), and the case-insensitive searches recorded in Section A find no occurrence of either name in the sources. The DAG runtime that Part II studies 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 yet made. The formal results are unaffected, since none of them mentions a particular component system, but a reader entitled to expect a harness that composes three named systems should know that it composes one.

9 Related work

The immediate antecedent is Banu [2], whose correspondence this Part tests on an independent system. The differences are worth naming. Banu validates with five compiler functors and one reference implementation, and reports certificate preservation rates across those targets; this Part validates with one implementation and reports which certificate kinds can be preserved in principle, with a separation theorem distinguishing them. Banu’s certificates carry theorem statements by construction because he wrote them; the certificates here do not, which is what forces Theorem 3.6. Banu’s own limitations section names the static framework, the single reference implementation, the restriction of certificate scope to structural invariants, a two-model escalation experiment and an 8B-parameter format-discipline ceiling on SWE-bench-lite. This Part adds a second, independently developed system and one negative structural result; it does not address the experimental limitations at all.

The ArchAgents framework [4] supplies the triple. This Part uses its shape and gives its own definitions of the category and the monoidal structure, because the operational questions asked here, in particular the status of Φ under morphisms and the fence order on receipts, are finer than the comparative framework needs.

Cao [3] argues that agentic systems are a restructuring of software engineering rather than an improvement of it, on a complexity-scaling argument: interaction paths among n components grow as $\Theta(2^n)$ against roughly constant human capacity to reason about them. Theorem 6.3 is a small piece of counter-pressure on the optimistic reading of that argument. If composing two agent applications creates ordering information that belongs to neither, then the composite is not understandable by understanding the parts, which is the same difficulty in categorical dress.

Zhou et al. [19] give the four pillars, Meng et al. [9] the enumerative harness taxonomy, and Pan et al. [18] the natural language harness study that Banu reads as evidence that harnesses are portable objects with algebraic structure. Banu’s reference list attributes that study to a different first author; the attribution used here follows the arXiv record. Trivedy [16] states the industry framing that the model holds the intelligence and the harness is what makes it useful.

For the compositional apparatus, Spivak’s operad of wiring diagrams [15] and its directed [13] and dynamical [17] refinements are Part III’s source and are cited here only for the shape of G . Fong and Spivak [5] is the general reference. Mac Lane [8] supplies the monoidal category definitions. Rutten [14] and Jacobs [6] are Part I’s sources. Ma et al. [7] supply the atomic-skills result Banu reads as operadic. That preprint has since been withdrawn by its authors over errors in its data, which is worth recording because it is one of the two empirical supports Banu offers for the operadic reading of the Skills pillar. Nothing proved here depends on it; it is cited only through Part II and only for the composition vocabulary.

Prior work by the present author describes a different harness in a different runtime. The `agent-os` papers at commit 61cb399 present an Elixir umbrella application composing a scheduler, a tool interface, a memory layer and a planner engine through a dependency-ordered supervision

tree, and argue that every pairwise combination of subsystems yields capabilities that neither has alone [1]. That is the same composition-creates-structure claim made informally, and Theorem 6.3 is one precise instance of it. A naming collision must be flagged: the **agent-os** series has its own Part IV, the Planner Engine, which is unrelated to this series' Part IV. Where this paper writes Part IV it always means the present one.

10 Conclusion

At commit 1c3ad24, application manifests, hook records, and name-keyed resolution make the three components of an Architecture triple identifiable. Five laws hold, deterministic replay identity is partial, and deployment-independent checking fails for managed artifact containment. The evidence therefore supports a structural correspondence, not a transfer of system guarantees: the modeled certificates attest identity and record form rather than substantive theorems.

11 Open problems

The harness needs a theorem field. Theorem 3.6 is a degenerate case of Theorem 3.3, and the gap between them is a specific piece of engineering: a claim language in which a hook can state what its receipt attests, and a checker that evaluates the claim against the receipt rather than against the request's identity fields. Until that exists, Theorem 3.32(1) preserves identity and nothing else.

The choice between one fence sequence and many remains open. Theorem 6.3 and Theorem 6.7 exhibit both designs in one system. The question is whether the global allocation order the durability barrier provides is worth the loss of strength. That order is not itself a commit or replay order; a replay claim requires evidence that consumers sort by the field and tolerate gaps and commit-order inversions.

Sys also needs to record effects that were performed but not recorded. Section 8.6 shows that Law 5 is about the store and not about the world, and that the model has no way to say so. An object of **Sys** carrying a set of external effect identifiers, with morphisms required to preserve them, would let the gap between a fenced record and an unfenced side effect be stated as a failure of a square to commute rather than as a remark.

The two disjointness conditions in Section 7.3 may be instances of one theorem. Part II's equivariance condition on sibling output keys and this Part's compatibility condition on global type identifiers are both partiality conditions on a composition law, arising from name collision in a shared namespace. If there is a single statement subsuming them, it would apply at every level of this series, and if there is not, the reason would explain why the levels are genuinely different.

A Code evidence

Every code-backed claim in the paper has a row. All **AgentHero** rows are at commit 1c3ad24; the **agent-os** row is at commit 61cb399. Pinning to an immutable commit hash is what makes the citations stable rather than fragile: a later refactor produces a different commit and leaves the cited one unchanged, so a reader can reconstruct exactly the artifact the claims were checked against. The formal results of Section 3 are independent of the code and remain whatever the repository does next; what a refactor can invalidate is the status of a law in Table 2, which is the intended, checkable content of Section 6. Statements in the last column are implementation-level claims established by source inspection at the stated commit, not reports of observed runs.

Two claims in Section 8 are absence claims and cannot be pinned to a line, since no line exhibits the absence of a name. They are stated here as the exact searches that establish them, each reproducible at the pinned commit. A case-insensitive recursive search of the **AgentHero** tree over files matching `*.rs`, `*.toml` and `*.mjs` for the strings `contextfs` and `catdb` returns no match. A separate recursive search over `*.rs`, `*.mjs` and `*.ts` for `RunContext`, `SkillOrganism`, `PatternTemplate`, `WiringDiagram`, `BiTemporalMemory` and `SkillStage`, the six component names in Banu’s four-pillar table, returns no match. The table below gives the positive, line-pinned evidence that neither system is declared as a dependency.

Claim	Repository, commit	Path and identifier	Lines	What the code shows
Know has two phases, admission and post-commit	AgentHero 1 c3ad24	<code>crates/orchestrator/src/hooks.rs</code> , <code>HookPhase</code>	37–45	The implementation declares <code>PrePolicy</code> as “deterministic admission policy evaluated before canonical mutation” and <code>PostCommit</code> as “side-effect-capable handler evaluated after a canonical event commits”
Each certificate declares its behaviour on failure	AgentHero 1 c3ad24	<code>crates/orchestrator/src/hooks.rs</code> , <code>HookFailurePolicy</code>	57–67	The implementation declares three variants, <code>Deny</code> , <code>Retry</code> and <code>Continue</code> , with <code>Deny</code> documented as denying the synchronous operation
Witnesses are closed records with a schema and identity fields (Theorem 3.6)	AgentHero 1 c3ad24	<code>crates/orchestrator/src/hooks.rs</code> , <code>HookReceipt</code>	427–445	The implementation declares a struct with <code>serde (deny_unknown_fields)</code> carrying <code>schema</code> , <code>status</code> , <code>plugin</code> , <code>hook_key</code> , <code>event_id</code> , <code>idempotency_key</code> and <code>output</code>
The replay check is total, decidable and identity-based (Law 2)	AgentHero 1 c3ad24	<code>crates/orchestrator/src/hooks.rs</code> , <code>validate_receipt</code>	693–727	The implementation compares six receipt fields against the request and tests that <code>output</code> is a JSON object, returning an error on any mismatch
Replay identity is a deterministic function of transported data (Law 3)	AgentHero 1 c3ad24	<code>crates/orchestrator/src/hooks.rs</code> , <code>hook_idempotency_key</code>	249–258	The implementation computes SHA-256 over a domain tag, the hook key and the event identifier with zero-byte separators
Pre-policy and post-commit identities are hashed from disjoint prefixes	AgentHero 1 c3ad24	<code>crates/orchestrator/src/hooks.rs</code> , <code>pre_policy_idempotency_key</code>	1002–1010	The implementation uses the distinct domain tag <code>agenthero.hook.pre-policy.v1</code> and a request identifier in place of an event identifier, so the two phases hash disjoint prefixes

Claim	Repository, commit	Path and identifier	Lines	What the code shows
Admission is fail-closed on three conditions (Law 4)	AgentHero 1 c3ad24	crates/orchestrator/src/hooks.rs, evaluate_pre_policy_hooks	1042–1145	The implementation raises when a plugin bundle hash is absent (line 1073, before any persistence), when the executor errors, and when the receipt fails validation, persisting a decision row in the latter two cases
Certificate scope is a pattern over events, applications and manifests	AgentHero 1 c3ad24	crates/orchestrator/src/hooks.rs, subscription query	1051–1066	The implementation selects enabled <code>pre_policy</code> rows by exact, wildcard or prefix event pattern and by optional application and manifest filters
Claims are leased and fenced before dispatch (Law 5)	AgentHero 1 c3ad24	crates/orchestrator/src/hooks.rs, claim_hook_delivery	551–639	The implementation selects one due row <code>forupdate...skiplocked</code> , increments <code>attempts</code> and <code>fence_token</code> , sets the lease, and commits before returning
A superseded writer cannot record a completion (Law 5)	AgentHero 1 c3ad24	crates/orchestrator/src/hooks.rs, complete_hook_delivery	642–662	The implementation predicates its update on <code>id</code> , <code>state</code> , <code>lease_owner</code> and <code>fence_token</code> and raises “stale hook completion was fenced” unless exactly one row changed
Failure handling is fenced and policy-directed	AgentHero 1 c3ad24	crates/orchestrator/src/hooks.rs, fail_hook_delivery	665–692	The implementation chooses <code>failed</code> or <code>dead_letter</code> from <code>HookFailurePolicy</code> and the attempt count, applies a capped quadratic backoff, and raises on a fence mismatch
Plugins run as environment-cleared child processes under a declared capability	AgentHero 1 c3ad24	crates/orchestrator/src/hooks.rs, InstalledPlugin HookExecutor::invoke	749, 1148–1180	The implementation checks the requested capability against the installed manifest, then spawns the resolved process with <code>env_clear()</code> at line 1163 and an explicit environment whitelist; no further sandboxing appears in the cited range
The durability barrier commits before dispatch under a lease	AgentHero 1 c3ad24	crates/orchestrator/src/durability.rs, module doc and commit_before_dispatch	1, 23–119	The module is documented as “canonical, lease-fenced commit-before-dispatch storage” and the function locks the worker lease, writes the node projection and inserts the event inside one transaction before committing

Claim	Repository, commit	Path and identifier	Lines	What the code shows
The durability fence is the global event identifier (Theorem 6.3)	AgentHero 1 c3ad24	crates/orchestrator/src/durability.rs, response_for	271–285	The implementation sets <code>fence:u64::try_from(event_id)?</code> and a durable reference naming the <code>dag_events</code> row
The event identifier comes from one sequence shared by all applications (Theorem 6.3)	AgentHero 1 c3ad24	supabase/migrations/20260522000002_app_runtime_tables.sql, <code>dag_events</code>	132–143	The table is declared with <code>idbigserialprimarykey</code> and no application scoping in its key; at this commit the cited path is a symbolic link to <code>agenthero/migrations/20260522000002_app_runtime_tables.sql</code> , and the line numbers are those of the link target
The single sequence is intentional and gappy per application	AgentHero 1 c3ad24	supabase/migrations/20260827000003_lock_free_dag_event_sequence.sql	1–6	The migration states that <code>dag_events.id</code> is allocated by a non-transactional sequence and preserves a strict order within every application run with gaps allowed
Hook delivery fences are per row, not global (Theorem 6.7)	AgentHero 1 c3ad24	supabase/migrations/20260824000002_plugin_hooks.sql, <code>hook_deliveries</code>	40–59	The table declares <code>fence_tokenbigintnotnulldefault0</code> per row and <code>unique(subscription_id,event_id)</code>
The receipt carried by a node attempt is typed and fenced	AgentHero 1 c3ad24	crates/dag-executor/src/lib.rs, <code>DurabilityBarrierReceipt</code>	138–157	The implementation declares a <code>deny_unknown_fields</code> struct carrying <code>schemata</code> , <code>durable_ref</code> , <code>dag_type</code> , <code>manifest_hash</code> , <code>node_id</code> , <code>attempt</code> and <code>fence</code>
Φ 's model half is a name-keyed resolution function	AgentHero 1 c3ad24	crates/llm-adapter/src/lib.rs, <code>provider_by_name</code> , <code>LLMProvider</code>	431–446, 86–100	The implementation matches a short provider name against four feature-gated arms and returns <code>LLMError::Provider</code> for any other name
Φ 's process half is a manifest-declared adapter	AgentHero 1 c3ad24	crates/orchestrator/src/dag_apps.rs, <code>AppAdapter</code>	541–574	The implementation declares a single <code>Process</code> variant carrying a command, runtime and host command lists, an environment whitelist and fixed arguments
Deployment is manifest-driven and application-agnostic	AgentHero 1 c3ad24	crates/orchestrator/src/dag_apps.rs, <code>AppManifest</code> ; <code>docs/agenthero-control-plane.md</code>	363–391; 1–5	The manifest declares slug, adapter, release, observability contract, actions and deployments; the documentation states that one registry-driven surface serves every installed application

Claim	Repository, commit	Path and identifier	Lines	What the code shows
Dispatch crosses a typed request and response boundary	AgentHero 1 c3ad24	crates/agent-runtime/ src/app_protocol.rs, AppAdapterRequest, AppAdapterResponse	211–240, 368–390	The implementation declares a protocol-marked request carrying application, action, manifest type, input, an idempotency key and an optional checkpoint, and a matching response
External effect deduplication is delegated to the adapter (outside Law 5)	AgentHero 1 c3ad24	crates/agent-runtime/ src/app_protocol.rs, idempotency_key	233–235	The field is documented as a “stable key adapters use to deduplicate retried external side effects”, placing the deduplication obligation on the adapter rather than on the platform
The registry decides tensor admissibility (Law 7, Theorem 3.21)	AgentHero 1 c3ad24	crates/orchestrator/src/ dag_apps.rs, validate_unique_dag_types	2442–2458	The implementation raises when two distinct application slugs declare the same manifest type identifier
The admissibility check runs inside the enumeration loader	AgentHero 1 c3ad24	crates/orchestrator/src/ dag_apps.rs, load_app_manifests_from_root, load_app_manifests, registered_dag_apps	1410–1427; 1405–1406; 2461–2481	The implementation calls <code>validate_unique_dag_types</code> at line 1425 before returning the manifest list; <code>load_app_manifests</code> (line 1405) delegates to this loader and <code>registered_dag_apps</code> (line 2461) calls <code>load_app_manifests</code>
One check in the harness reads Φ (Law 6 violated)	AgentHero 1 c3ad24	packages/ harness-runtime/src/ workspace.mjs, resolveManagedArtifactPath	22–65	The implementation builds its authorized root set from the workspace root and <code>AGENTHERO_ARTIFACT_STORE_DIR</code> , refuses non-descendant paths and symbolic links, and separately compares a content hash carried in the artifact metadata
The workspace root is derived, not configured per node	AgentHero 1 c3ad24	packages/ harness-runtime/src/ workspace.mjs, workspaceRootFromOutput	10–19	The implementation walks upward from a scoped output directory until it finds a managed work directory
Provider output is normalized at the harness boundary	AgentHero 1 c3ad24	packages/ harness-runtime/src/ provider-session.mjs, budget constants, sanitizeProviderSessionEvents, ProviderSessionCapture	1–3, 8–111, 114–150	The implementation declares three explicit size budgets and converts provider JSONL into events the source describes as bounded and content-safe, suppressing repeated tool call identifiers

Claim	Repository, commit	Path and identifier	Lines	What the code shows
The harness carries a versioned node-control sub-protocol	AgentHero 1 c3ad24	packages/ harness-runtime/src/ provider-session.mjs, protocol constants, watchSchedNodeControls	4–5, 199–287	The file declares the versioned protocol identifiers <code>agenthero.node-control.v1</code> and <code>agenthero.node-controls.v1</code> , and the function requires a token and a run, DAG, node and attempt identity, polls the control feed, and converts a matching stop into an abort signal
The record half of the cross-language contract is generated from one registry	AgentHero 1 c3ad24	crates/ control-plane-contract/ src/lib.rs, AppRunEvent, TYPE_CONTRACTS, typescript_contract	20–45; 572; 1262–1269	The generator loops over the <code>TYPE_CONTRACTS</code> registry and emits one TypeScript type per entry from the Rust field list
The command half is written out by hand in the same generator	AgentHero 1 c3ad24	crates/ control-plane-contract/ src/lib.rs, AgentTeamCommand, typescript_contract	300–320; 1271–1290	The Rust enum is declared once, but its TypeScript union is emitted from a literal string in the generator rather than from the enum, so the two can drift
Per-operation policy is declared inside the manifest and is not classified as a certificate	AgentHero 1 c3ad24	crates/dag-runtime/src/ lib.rs, DagToolPolicy, DagTool.policy, DagManifest.tools	564–579; 506; 669	<code>DagToolPolicy</code> declares budget, approval, network, filesystem, subprocess and credential policy; it is reached only as the <code>policy</code> field of a <code>DagTool</code> , and the tool list is a field of the manifest
The harness-level wiring is the application manifest	AgentHero 1 c3ad24	crates/dag-runtime/src/ lib.rs, DagManifest, DagManifest::validate, DagEdge	657–676; 684; 636–646	The implementation declares a manifest carrying identity, version, tools, roles, nodes and edges, and a <code>validate</code> method on it
The repository describes itself as the platform, not one pillar	AgentHero 1 c3ad24	CLAUDE.md, “Product Boundary”; README.md, “Platform Extensions”	1–12; 32–43	The documentation states that the platform owns the control plane, discovery, executor contracts, runtime state, scheduling and adapter dispatch, and that pre-policy hooks fail closed while post-commit hooks use durable attempts, leases, fencing, idempotency and retained receipts
The harness declares no dependency on the memory or protocol systems (Section 8.7)	AgentHero 1 c3ad24	Cargo.toml workspace members and dependencies; packages/ harness-runtime/package.json	1–72; 1–21	The workspace lists eight member crates and its third-party dependency set, none naming <code>ContextFS</code> or <code>CatDB</code> ; the harness runtime package declares no dependency block at all

Claim	Repository, commit	Path and identifier	Lines	What the code shows
Session content is projected into the store as a flat, event-keyed table (Section 7.2)	AgentHero 1 c3ad24	supabase/migrations/ 20260824000001_agent_ session_memory.sql, agent_ _session_memory_chunks	4–31	The full table declaration keys each chunk by <code>source_event_id</code> and carries application, DAG, node, provider, model and session columns; no column of the declaration names a relation to another chunk
A prior harness by the same author composes four subsystems in a different runtime	agent-os 61c b399	papers/latex/synthesis. tex, abstract; repository root agentherowork/agent-os	117–148	The abstract states that an Elixir umbrella application composes a scheduler, tool interface, memory layer and planner engine through a dependency-ordered supervision tree, and that every pairwise combination yields capabilities neither has alone

References

- [1] M. Long. The AI Operating System, Part V: Modular Synthesis. `agent-os` repository, `papers/latex/synthesis.tex`, commit 61cb399, 2025.
- [2] B. Banu. Harness Engineering as Categorical Architecture: Structural Guarantees Are Harness-Level Properties. [arXiv:2605.12239](https://arxiv.org/abs/2605.12239), 2026.
- [3] Z. Cao. Agentic Software: How AI Agents Are Restructuring the Software Paradigm. [arXiv:2606.05608](https://arxiv.org/abs/2606.05608), 2026. The version consulted here is titled “The End of Software Engineering: How AI Agents Are Fundamentally Restructuring the Software Paradigm”.
- [4] P. de los Riscos, F. J. Corbacho, and M. A. Arbib. Towards a category-theoretic comparative framework for artificial general intelligence. [arXiv:2603.28906](https://arxiv.org/abs/2603.28906), 2026.
- [5] B. Fong and D. I. Spivak. Seven Sketches in Compositionality: An Invitation to Applied Category Theory. [arXiv:1803.05316](https://arxiv.org/abs/1803.05316), 2018.
- [6] B. Jacobs. Introduction to Coalgebra: Towards Mathematics of States and Observation. Cambridge Tracts in Theoretical Computer Science 59, Cambridge University Press, 2016.
- [7] Y. Ma, Y. Liu, X. Yang, et al. Scaling coding agents via atomic skills. [arXiv:2604.05013](https://arxiv.org/abs/2604.05013), 2026. Withdrawn by the authors; the submission history records errors in the data affecting the validity of the results.
- [8] S. Mac Lane. Categories for the Working Mathematician. Graduate Texts in Mathematics 5, Springer, 2nd edition, 1998.
- [9] Q. Meng, Y. Wang, L. Chen, Q. Wang, C. Lu, W. Wu, Y. Gao, Y. Wu, and Y. Hu. Agent harness for large language model agents: A survey. Preprints, DOI 10.20944/preprints202604.0428.v2, 2026.

- [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] D. Rupel and D. I. Spivak. The operad of temporal wiring diagrams: Formalizing a graphical language for discrete-time processes. arXiv:1307.6894, 2013.
- [14] 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.
- [15] D. I. Spivak. The operad of wiring diagrams: Formalizing a graphical language for databases, recursion, and plug-and-play circuits. arXiv:1305.0297, 2013.
- [16] V. Trivedy. The anatomy of an agent harness. LangChain Blog, March 2026. <https://www.langchain.com/blog/the-anatomy-of-an-agent-harness>.
- [17] 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.
- [18] L. Pan, L. Zou, S. Guo, J. Ni, and H.-T. Zheng. Natural-language agent harnesses. arXiv:2603.25723, 2026. Banu’s reference list attributes this work to “Erik Willstrom et al.”; the arXiv record lists the authors given here.
- [19] 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.