

Typed Protocol Wiring

Part III of *Agentic Engineering*

Matthew Long

The YonedaAI Collaboration, YonedaAI Research Collective

Chicago, IL

`matthew@yonedaai.com`

September 2026

Abstract

The categorical reading of agent architecture assigns the protocol layer to the syntactic wiring G of an architecture triple (G, Know, Φ) . We make it precise and test it against one system. A protocol is a typed box carrying two further pieces of boundary data: a refusal-complete outcome structure on each output port, in which a successful empty answer and each refusal are distinct values, and a locus map sending each refusal to a published name. Protocols form an algebra over the operad of strict tree wiring diagrams, so refusals compose along substitution without being flattened, and a counterexample shows that the tree restriction cannot be dropped. Six realisation conditions are checked against CatDB at commit `7cc9341a`, an entity-federation database with a Model Context Protocol tool surface. Five hold on specified fragments; encoding discrimination is not established. The principal result is a confinement theorem for the query algebra: in every plan the implementation accepts, the field names reaching the caller lie inside the names introduced at the plan's leaves. Two of the eight node cases carry the induction, and the crate establishing the result cannot state the leaf hypothesis under which it would bear on authorization. The limits are structural: wiring and policy share a crate, the schema crate is a placeholder, port types encode names rather than value types, and a tool wired into a manifest-driven runtime loses the port discipline. The evidence is source inspection at a pinned commit, not execution.

Contents

1	Introduction	2
1.1	Contributions	3
1.2	Scope and dependencies	3
2	Background	4
2.1	The protocol pillar	4
2.2	The architecture triple	4
2.3	Wiring diagrams as an operad	5
2.4	Relational plans	5
3	The formal model	5
3.1	Port types and boxes	5
3.2	The wiring operad	6

3.3	Outcomes, refusals, loci	8
3.4	Protocol composition	9
3.5	Realisation conditions	12
3.6	Model faithfulness	13
4	The system	13
4.1	Mode-indexed tool surface	14
4.2	The wire format	15
4.3	Plan algebra	15
4.4	Typed outcomes and refusals	16
4.5	Federation refusal lifting	17
4.6	Provenance and policy	17
4.7	Transport surfaces	18
4.8	Connector conformance	19
4.9	Mapping table	20
5	Law checking	20
5.1	W1, port compatibility	20
5.2	W2, hereditary checking	20
5.3	W3, leaf confinement	20
5.4	A linearity condition on unions	22
5.5	W4, refusal completeness	23
5.6	W5, boundary honesty	23
5.7	W6, encoding discrimination	24
5.8	Audit summary	24
6	Layer composition	24
6.1	Layer contribution	24
6.2	Operad distinction	25
6.3	Cross-layer scope	26
7	Limits and counterexamples	26
7.1	Coupled wiring and policy	26
7.2	Schema crate stub	27
7.3	Nominal port labels	27
7.4	Tree-fragment implementation	28
7.5	Threats to validity	28
8	Related work	29
9	Conclusion	29
A	Code evidence	30

1 Introduction

An agent that calls a database does two things at once. It composes a program out of typed parts, and it negotiates with a system that may decline. The first activity has a mature mathematics behind it. The second rarely appears in that mathematics, because the standard formalisms for

composition treat failure as an afterthought: a partial function, an exception, an error string appended to a return type. The question taken up here is what the compositional account becomes when declining is treated as first-class data on a port.

The setting is the externalization programme for language-model agents. Zhou et al. [15] organise the components that sit outside the model into four pillars: memory, skills, protocols, and harness. Banu [1] argues that these four already have a formal home in the ArchAgents architecture triple (G, Know, Φ) of de los Riscos, Corbacho and Arbib [9]. Memory becomes coalgebraic state, skills become operations of an operad, protocols become the syntactic wiring G with typed ports, and the harness is the whole triple. The correspondence is attractive and it is largely untested against code written for other reasons. The system examined below is **CatDB**, a federating entity database whose agent-facing surface is a Model Context Protocol [7] tool set, read at commit `7cc9341a`.

Two questions organise the work. The first is whether **CatDB**’s typed-outcome and refusal surface constitutes a syntactic wiring G with typed ports in the sense of Spivak’s operad of wiring diagrams [11, 10, 12], and which port-type laws its planning and policy layers enforce. The second is what a protocol is here that a typed software interface is not.

The second question carries most of the weight. Every serialisation format has typed ports in a trivial sense, and a reading of the protocol pillar that amounts to observing that requests and responses have types is a restatement rather than a theory. The answer given below is that a protocol is a typed box together with two further pieces of boundary data. The first is a refusal-complete outcome structure on each output port: an outcome in which a successful empty answer and each way of declining are distinct observable values rather than being collapsed into one. The second is a locus map, which sends each way of declining to a name the box publishes, so that a refusal raised inside a composite still names its origin when it reaches the boundary. A typed interface constrains what may be said. A protocol also constrains what may be said when nothing can be said, and constrains it in a way that survives composition, which is the content of Theorem 3.18.

1.1 Contributions

The paper defines protocols as typed boxes whose boundary data distinguish successful empty values from typed refusals and preserve each refusal’s locus. On output-supported strict tree wirings, these protocols admit an operad action by disjointly accumulating upstream reasons (Theorem 3.18). Six conditions separate diagram validity, protocol behaviour, deployment, and encoding (Section 3.5).

The source audit proves field confinement for the eight-node **CatDB** query algebra: every field available at a validated plan’s root was introduced by a scan leaf (Theorem 5.2). It also locates the model’s boundary. The implementation uses field names rather than base or value types, its schema crate is a placeholder, and the manifest layer above it performs no producer–consumer schema comparison. Each code-backed claim is tied to an identifier at a pinned commit in Section A; the audit does not claim execution or mechanised verification.

1.2 Scope and dependencies

The four-pillar correspondence is examined here on one pillar and one system. Nothing below claims that the pillars and the architecture triple are the same thing. The claim is that under stated assumptions there is a structure-preserving interpretation of the protocol pillar in the wiring component G , that **CatDB** realises it on an identifiable fragment, and that the fragment has a boundary, given in Section 7.

The paper is self-contained. Every definition used below is given here or taken from published work cited at its first use. The architecture triple comes from de los Riscos, Corbacho and Arbib [9],

the certificate notion from Banu [1], and the wiring operad from Spivak and successors [11, 10, 12, 14]. Companion work treats the other three pillars in detail [16, 17, 18]. It is cited for context, and no argument below rests on it. All claims about **AgentHero**, which appears in Section 6 as a contrasting case, are established here by inspection of the files named in Section A.

Every statement about **CatDB** or **AgentHero** below is an implementation-level claim established by reading the source at a pinned commit. To say that the implementation at commit `7cc9341a` refuses a plan is to say that the function named in the evidence table returns an error value on that input. That is read off the body of the function; no running deployment was observed doing it. No property here is verified in the sense of machine-checked proof, and no word of that strength is applied to anything the source merely performs. Theorem 5.2 is a mathematical statement about a model of the plan type, proved below. Its bearing on the implementation is exactly as strong as the faithfulness of that model, which Section 3.6 sets out.

2 Background

2.1 The protocol pillar

Zhou et al. [15] survey the externalization of agent capability into components the model does not contain. Their protocol pillar covers the interface conventions by which an agent addresses a tool or another agent: the schema of a call, the shape of a reply, the discipline that lets one agent’s output be another’s input. Meng et al. [8] give an enumerative alternative, a six-component taxonomy of harness features, against which Banu contrasts the categorical reading.

Banu [1], Section 3.3, states the categorical reading we test. Protocols are the syntactic wiring G : typed input and output ports with compatibility checks between them, and wires carrying integrity labels drawn from a small set (validated, raw, sanitized). Three optical shapes sit at the wire level, lenses for constitutional access, prisms for conditional routing, and traversals for batch processing. Banu’s own reference implementation names a **WiringDiagram** type. None of Banu’s reference identifiers occur in the systems this series examines. Those systems independently realise an analogous structure, and are not described here as implementing Banu’s vocabulary.

Cao [2] supplies the motivation from the other direction. His Proposition 2.1 states the number of possible interaction paths as $P(n) \in \Theta(2^n)$. The derivation counts dependency graphs on n labelled components and therefore yields $2^{\binom{n}{2}} = 2^{\Theta(n^2)}$; the latter expression is this series’ correction, not Cao’s stated rate. The human capacity to reason about those interactions stays roughly constant. If that is why capability is being pushed out of code and into runtime-composed tooling, then the interface between agent and tool is the surface across which the complexity is being moved, and the discipline that surface carries is worth knowing.

2.2 The architecture triple

An architecture in the sense of de los Riscos, Corbacho and Arbib [9], as relayed by Banu, is a triple $A = (G_A, \text{Know}_A, \Phi_A)$. Here G_A is a syntactic wiring, a graph of modules with ports and directed edges; Know_A is a knowledge structure carrying the structural properties and certificates that hold of G_A ; and Φ_A is a deployment map from abstract capability slots to concrete implementations. A morphism of architectures is a structure-preserving translation, in practice a compiler. One further notion is needed: what populates Know . Banu [1], Definition 1, takes a certificate to be a triple $(\tau, \sigma, \text{evds})$. In it τ is a statement, σ maps the symbols of τ to parameters of the architecture, and evds is a derivation that can be mechanically replayed to check that τ holds. That definition is used below only to say what the objects found in **CatDB** are not. Nothing in the sequel depends on any

property of the triple beyond what the two citations just made supply. What is supplied here is the G component and an account of where its boundary with Know falls in one system.

2.3 Wiring diagrams as an operad

The formal apparatus is Spivak’s operad of wiring diagrams [11] and its directed and typed refinements [10, 12]. A diagram of boxes connected by wires is not merely a picture but an operation. Given n inner boxes and one outer box, a wiring diagram says how to build the outer behaviour from the inner ones, and diagrams compose by substituting one diagram for an inner box of another. That composition is associative and unital, which is exactly the statement that wiring diagrams form a coloured operad whose colours are boxes. Vagner, Spivak and Lerman [12] then study algebras over this operad, which assign to each box a set of behaviours and to each diagram a function combining them. Fong and Spivak [4] give an accessible treatment of the surrounding compositional vocabulary.

We recall the construction in Section 3 in a form adapted to the present purpose, with two departures from the sources. Port types carry a subtyping preorder rather than equality, because the concrete port types below are finite sets of field names ordered by reverse inclusion. The tree fragment is then isolated, because the system under study represents its wiring as an owned tree and therefore cannot express sharing.

2.4 Relational plans

CatDB’s interior wiring is a relational operator tree in the tradition that begins with Codd [3]. We use no result from that tradition, only its vocabulary: scan, project, filter, join, union, aggregate, sort, limit. Three things about that tree matter here. It is already a typed wiring diagram in everything but name. The implementation checks port compatibility at every node rather than only at the root. And the check has a consequence about authorization that the implementation does not itself state.

3 The formal model

3.1 Port types and boxes

Definition 3.1 (Type alphabet). A *type alphabet* is a pair $(\mathsf{T}, \leq)$ where T is a set of *port types* and $\leq$ is a preorder on T , the *substitutability* order, read $s \leq t$ as “a supply of type s satisfies a demand of type t ”. The alphabet is *discrete* when $\leq$ is equality. A *typed port* over T is a pair $p = (n_p, \text{ty}(p))$ of a name and a type.

The substitutability order is not decoration. The concrete alphabet of Section 4 is $\mathsf{T} = \mathcal{P}_{\text{fin}}(\text{Fld})$, the finite sets of field names, with $S \leq D$ if and only if $S \supseteq D$: a wire carrying more fields than are demanded is acceptable, one carrying fewer is not. This is width subtyping for records, and every port-compatibility statement below is relative to it.

Definition 3.2 (Box). A *box* over T is a pair $X = (X^{\text{in}}, X^{\text{out}})$ of finite sets of typed ports, with distinct names inside each set. A *box map* $u: X \rightarrow X'$ is a pair of injections $X^{\text{in}} \hookrightarrow X'^{\text{in}}$ and $X^{\text{out}} \hookrightarrow X'^{\text{out}}$ preserving types.

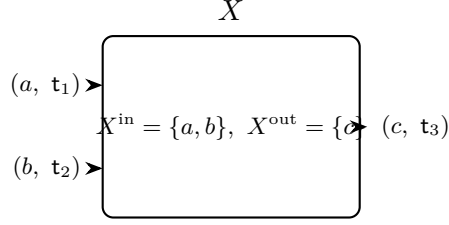


Figure 1: A box with two typed input ports and one typed output port. Boxes are the colours of the operad $\mathcal{W}_T$; wiring diagrams are its operations.

3.2 The wiring operad

Fix a type alphabet $(T, \leq)$. Given boxes $X_1, \dots, X_n$ (the *inner* boxes) and Y (the *outer* box), write

$$\text{Dem}(\vec{X}; Y) = \left(\prod_{i=1}^n X_i^{\text{in}} \right) \sqcup Y^{\text{out}}, \quad \text{Sup}(\vec{X}; Y) = \left(\prod_{i=1}^n X_i^{\text{out}} \right) \sqcup Y^{\text{in}}.$$

The *demand* ports are the ones that need a value: an inner box's inputs, and the outer box's outputs, which must be filled before the composite can answer. The *supply* ports are the ones that provide a value: an inner box's outputs, and the outer box's inputs.

Definition 3.3 (Wiring diagram, the operad $\mathcal{W}_T$). A *wiring diagram* $\varphi: (X_1, \dots, X_n) \rightarrow Y$ is a function $\varphi: \text{Dem}(\vec{X}; Y) \rightarrow \text{Sup}(\vec{X}; Y)$ such that

- (i) $\text{ty}(\varphi(p)) \leq \text{ty}(p)$ for every demand port p , and
- (ii) the relation $\prec$ on $\{1, \dots, n\}$ given by $i \prec j$ whenever some port of X_j^{in} is sent by φ to a port of X_i^{out} is acyclic.

Let $\mathcal{W}_T(X_1, \dots, X_n; Y)$ be the set of such diagrams. The identity $\text{id}_X \in \mathcal{W}_T(X; X)$ sends each port of the inner X^{in} to the like-named port of the outer X^{in} . It sends each port of the outer X^{out} to the like-named port of the inner X^{out} .

Substitution needs care, because the ports of the box being replaced change role. Let $\varphi \in \mathcal{W}_T(X_1, \dots, X_n; Y)$ and $\psi \in \mathcal{W}_T(Z_1, \dots, Z_m; X_i)$. Write

$$\vec{X}' = (X_1, \dots, X_{i-1}, Z_1, \dots, Z_m, X_{i+1}, \dots, X_n)$$

for the substituted list. The ports of X_i^{in} are demands of φ and supplies of ψ ; the ports of X_i^{out} have the opposite roles. Neither family belongs to $\text{Dem}(\vec{X}'; Y)$ or $\text{Sup}(\vec{X}'; Y)$, so a composite demand is resolved by a walk that may change level.

Definition 3.4 (Substitution). For $d \in \text{Dem}(\vec{X}'; Y)$ define a sequence $d_0 = d$ and

$$d_{k+1} = \begin{cases} \varphi(d_k) & \text{if } d_k \in \text{Dem}(\vec{X}; Y), \\ \psi(d_k) & \text{if } d_k \in \text{Dem}(\vec{Z}; X_i), \end{cases}$$

stopping at the first k with $d_k \in \text{Sup}(\vec{X}'; Y)$. Set $(\varphi \circ_i \psi)(d) = d_k$.

Lemma 3.5 (Termination and boundedness). *The sequence of Theorem 3.4 reaches the set $\text{Sup}(\vec{X}'; Y)$ after three steps at most. Hence $\varphi \circ_i \psi$ is a well-defined function, and it satisfies conditions (i) and (ii) of Theorem 3.3.*

Proof. There are two kinds of starting demand. If $d \in Z_j^{\text{in}}$ then $d_1 = \psi(d)$ lies in $\coprod_{j'} Z_{j'}^{\text{out}}$, which is a composite supply and stops the walk, or in X_i^{in} , in which case $d_2 = \varphi(d_1)$ lies in Y^{in} or in $X_{i'}^{\text{out}}$ for some i' . The case $i' = i$ is excluded, since it would give $i \prec i$ in φ , contradicting acyclicity; every other value is a composite supply. If $d \in X_{i'}^{\text{in}}$ for $i' \neq i$ the same argument applies with the first step taken by φ . If $d \in Y^{\text{out}}$ then $d_1 = \varphi(d)$ is a composite supply unless it lies in X_i^{out} , in which case $d_2 = \psi(d_1)$ is a composite supply unless it lies in X_i^{in} , and then $d_3 = \varphi(d_2)$ is a composite supply by the argument just given. So the walk stops after at most three steps and no unbounded alternation between the two levels occurs. Condition (i) holds because each step is $\leq$ -decreasing and $\leq$ is transitive. For condition (ii), a cycle in the composite would contract, on replacing every Z_j by X_i , to a closed walk in $\prec_\varphi$, which is acyclic. So the cycle lies inside the Z_j , which makes it a cycle in $\prec_\psi$, also acyclic. $\square$

Proposition 3.6. $\mathcal{W}_\top$ with the data of Theorem 3.3 is a coloured symmetric operad, with boxes as colours and permutation of the inner boxes as the symmetric action.

Proof. Substitution is well defined and preserves conditions (i) and (ii) by Theorem 3.5. Unitality is immediate from the definition of id_X , whose walk has length one at every demand. Associativity is checked pointwise on demand ports. Both $(\varphi \circ_i \psi) \circ_j \chi$ and $\varphi \circ_i (\psi \circ_{j'} \chi)$ resolve a demand by the walk of Theorem 3.4 taken over all three levels, and the walk is deterministic, so the two resolutions agree. The case of substitution into two distinct inner boxes is the parallel version of the same argument. The symmetric action is relabelling of the inner index set, which commutes with substitution because the walk never consults the index. The construction is the typed directed case of [11]; see [10] for the directed variant, [12] for the version with typed ports used here, and [14] for a systematic treatment. $\square$

Write $\text{Box}_\top^1$ for the collection of boxes with exactly one output port.

Definition 3.7 (Tree wiring). Let $X_1, \dots, X_n, Y \in \text{Box}_\top^1$. A diagram $\varphi \in \mathcal{W}_\top(\vec{X}; Y)$ is a *tree wiring* if

(T1) φ is injective, so no supply port is sent to by two distinct demand ports, and

(T2) every port of $\coprod_i X_i^{\text{out}}$ lies in the image of φ , so no inner box is dead.

Write $\mathcal{W}_\top^{\text{tr}}(\vec{X}; Y)$ for the set of tree wirings.

Condition (T1) forbids fan-out: a value produced once is consumed once. Condition (T2) forbids dead boxes. Rootedness is not assumed; it follows.

Lemma 3.8 (Tree wirings are rooted at Y). *Let $\varphi \in \mathcal{W}_\top^{\text{tr}}(\vec{X}; Y)$. Then every inner box is upstream of the unique output port q of Y . The graph whose vertices are the inner boxes together with Y , with an edge from X_i to the box owning the demand that consumes X_i 's output, is a tree rooted at Y .*

Proof. Each box has exactly one output port. By (T2) that port is consumed by some demand, and by (T1) by exactly one, so the described graph is well defined and every inner vertex has out-degree one. Follow the edges from any inner box. The walk cannot repeat a vertex, since a repeat would give a cycle in $\prec$, excluded by Theorem 3.3(ii). There are finitely many boxes, so the walk terminates, and it can terminate only at a vertex with no outgoing edge. The only such vertex is Y , because every inner box has out-degree one. Hence every inner box is connected to Y by a directed path, and the graph is connected with a unique sink. A finite connected digraph in which every vertex but one has out-degree one and there are no cycles is a tree rooted at that vertex. Ports of Y^{in} are supplies and may be unused. That does not affect the argument, which concerns boxes. $\square$

So a tree wiring is a term in an algebraic signature, with Y at the root, and no separate rootedness clause is needed. The awkward case, several disconnected components flowing into Y^{in} , is excluded because Y^{in} carries supplies rather than demands and because $\text{Box}_\top^1$ forces Y^{out} to be a single port.

Proposition 3.9. $\mathcal{W}_\top^{\text{tr}}$ is a sub-operad of $\mathcal{W}_\top$ on the colours $\text{Box}_\top^1$.

Proof. The identity satisfies (T1) and (T2) since it is a bijection. For (T1) under substitution, suppose two distinct composite demands $d \neq d'$ resolve to the same supply t . By Theorem 3.5 each resolution is a walk of length at most three, and by injectivity of φ and of ψ separately, a walk is determined backwards by its endpoint together with the level at which the last step was taken. The last step of both walks is taken by the same map, since t determines whether it lies in $\coprod_j Z_j^{\text{out}}$, in $\coprod_{i' \neq i} X_i^{\text{out}}$ or in Y^{in} . So the two walks agree at their penultimate ports, and inductively at their starting points, giving $d = d'$. For (T2), let $t \in Z_j^{\text{out}}$. By (T2) for ψ , $t = \psi(d)$ for some $d \in \text{Dem}(\vec{Z}; X_i)$. If $d \in Z_{j'}^{\text{in}}$ then d is a composite demand resolving to t in one step. Otherwise $d \in X_i^{\text{out}}$, and by (T2) for φ there is e with $\varphi(e) = d$. Acyclicity of φ excludes $e \in X_i^{\text{in}}$, so e is a composite demand and its walk resolves to t in two steps. The case $t \in X_{i'}^{\text{out}}$, $i' \neq i$, is (T2) for φ directly. $\square$

Section 4 shows that CatDB’s plan representation maps into $\mathcal{W}_\top^{\text{tr}}$, and Section 7 discusses what that costs.

3.3 Outcomes, refusals, loci

The structure that separates a protocol from a typed interface is added to boxes, not to diagrams. This is deliberate. Boxes are the colours of $\mathcal{W}_\top$ and diagrams are its operations, so attaching data to a diagram would make a compositional boundary depend on the internals it is meant to hide. Everything in this subsection is boundary data: it is visible at a box’s ports and says nothing about how the box is built.

Definition 3.10 (Outcome structure). An *outcome structure* is a tuple $\omega = (V, V^\varnothing, R, O, \text{obs})$ in which V is a set of values, $V^\varnothing \subseteq V$ is a distinguished subset of *empty* values, R is a finite set of *reasons*, O is the set of values a caller receives, and

$$\text{obs}: V \amalg R \longrightarrow O$$

is the *observation map* out of the coproduct. The structure is *refusal-complete* when the restriction of obs to $V^\varnothing \amalg R$ is injective, and *nondegenerate* when $R \neq \emptyset$.

The subset $V^\varnothing$ is part of the data rather than something read off V . Which values count as an answer that succeeded and carried nothing is a modelling decision that the type alone does not settle. For a query port, $V^\varnothing$ is the singleton containing the empty row set; for a probe port it is the outcome of a probe that examined no candidates.

The mathematical content of Theorem 3.10 is entirely in the observation map. In **Set** a coproduct always has disjoint injections, so writing $V \amalg R$ guarantees nothing. What a caller can distinguish is determined by obs . A return type of `Result<Vec<Row>, ()>` whose error case is reported to the caller as an empty vector has obs collapsing $V^\varnothing$ and R , and is not refusal-complete however carefully the internal type is written. A return type of `Result<Vec<Row>, String>` in which every failure formats to the same message is not refusal-complete either, because obs is not injective on R . This is the formal content of the design principle CatDB’s README states as “a failure never looks like an empty result”.

A refusal is only actionable if it says what it is about. The place it points to must also be boundary data. We fix one ambient alphabet of names, shared by every box, rather than giving each box its own. A per-box name set would be carried through composition. It would leave the names of an intermediate box stranded in the composite after that box is substituted away, and that is exactly the data that must not survive substitution.

Definition 3.11 (Name alphabet, locus map). Fix a set $\mathbf{Nm}$ of *names*, containing the names of all ports of all boxes under consideration. A *locus map* for an output port q with reason set R_q is a function $\ell_q: R_q \rightarrow \mathbf{Nm}$.

Only the set structure of $\mathbf{Nm}$ is used below, and nothing requires its elements to be atomic. In the system of Section 4 the loci are structured values, a `JoinPath` or an `ArrowName`, and the model takes them as elements of $\mathbf{Nm}$ as they are. What composition needs is that a locus is boundary data surviving substitution unchanged, not that it is a bare string.

Definition 3.12 (Protocol). A *protocol* is a tuple

$$\mathcal{P} = (X, \{\omega_q\}_{q \in X^{\text{out}}}, \{\ell_q\}_{q \in X^{\text{out}}})$$

consisting of a box X over $\mathbf{T}$, a refusal-complete outcome structure ω_q on each output port, and a locus map ℓ_q for each. The *underlying typed interface* of $\mathcal{P}$ is the box X . Two protocols on the same box are *isomorphic* when there are bijections between corresponding reason sets and observation codomains that commute with the observation and locus maps and act as the identity on values. Write $A(X)$ for the set of isomorphism classes of protocols on X .

Remark 3.13. A locus map says what a refusal is about, not which box it came from. The second question is answered by the reason set itself, which is a coproduct after composition, so a composite refusal carries the index of the port that raised it. Keeping the two separate is what makes composition strictly associative, and it is what the implementation of Listing 6 does. The index is a `SourceId` on the report line and the name is inside the source's own typed reason.

Remark 3.14. The difference between a protocol and a typed interface is not one of expressive power. Any protocol can be encoded in a sufficiently rich interface type, and any interface can be given a one-element reason set. The difference is what the formalism forces one to say. A typed interface fixes what a well-formed request and a well-formed answer look like. A protocol additionally fixes how many ways there are of declining, requires each to be distinguishable from a successful empty answer and from the others, and requires each to name something. Those requirements are what make the failure behaviour compositional, which is the content of Theorem 3.16 and Theorem 3.18.

3.4 Protocol composition

Composition of protocols is composition along a diagram. We restrict the action to diagrams that are strict at the outer output ports and whose outer outputs are supplied by inner outputs. The second restriction excludes nullary passthroughs, for which the construction has no inner protocol from which to obtain outcome data.

Definition 3.15 (Strict diagram). A diagram $\varphi \in \mathcal{W}_{\mathbf{T}}(\vec{X}; Y)$ is *strict* if $\text{ty}(\varphi(q)) = \text{ty}(q)$ for every $q \in Y^{\text{out}}$. Strict diagrams are closed under substitution and contain the identities, so they form a sub-operad $\mathcal{W}_{\mathbf{T}}^s \subseteq \mathcal{W}_{\mathbf{T}}$.

Strictness constrains the outer output ports only. Interior wires and outer input ports retain the substitutability preorder of Theorem 3.3(i). The model specifies no coercion between the value

sets carried by comparable port types, so it cannot soundly retype an inner outcome at an outer boundary. Equality avoids inventing such a coercion. In the concrete alphabet of Section 4, a final **Project** can make a non-strict outer supply strict by narrowing its field-name set.

Fix $n \geq 1$, an output-supported $\varphi \in \mathcal{W}_T^s(X_1, \dots, X_n; Y)$, and protocols $\mathcal{P}_i$ on the X_i , where output-supported means $\varphi(Y^{\text{out}}) \subseteq \coprod_i X_i^{\text{out}}$. The index set of the composite reason set has to be a set of ports, not a set of boxes. A box with two output ports may refuse in different ways at each, and collecting both under one port would already break the unit law. Define $\text{up}: \text{Sup}(\vec{X}; Y) \rightarrow \mathcal{P}(\coprod_i X_i^{\text{out}})$ by recursion on the acyclic order $\prec$ of Theorem 3.3:

$$\text{up}(y) = \emptyset \quad (y \in Y^{\text{in}}), \quad \text{up}(p) = \{p\} \cup \bigcup_{d \in X_i^{\text{in}}} \text{up}(\varphi(d)) \quad (p \in X_i^{\text{out}}). \quad (1)$$

This is the boundary-level reading of dependence. A box's outputs are taken to depend on all of its inputs, because a box's internals are not visible. For $q \in Y^{\text{out}}$ put $P(q) = \text{up}(\varphi(q))$ and define

$$V_q = V_{\varphi(q)}, \quad V_q^\emptyset = V_{\varphi(q)}^\emptyset, \quad R_q = \coprod_{p \in P(q)} R_p, \quad (2)$$

$$\ell_q = [\ell_p]_{p \in P(q)}: R_q \longrightarrow \mathbf{Nm}, \quad (3)$$

where R_p and ℓ_p are the reason set and locus map of the protocol on the box owning p , and (3) is the copairing of those maps out of the coproduct. Put $p_0 = \varphi(q)$. Let obs_q use obs_{p_0} on $V_q \amalg R_{p_0}$ and $\text{obs}_p|_{R_p}$ on every other reason summand, with codomain

$$O_q = O_{p_0} \amalg \coprod_{p \in P(q) \setminus \{p_0\}} \text{obs}_p(R_p).$$

Write $A(\varphi)(\mathcal{P}_1, \dots, \mathcal{P}_n)$ for the resulting tuple on Y . No name set is carried, so nothing about X_i survives its own substitution except through the reason sets of the boxes that replaced it.

Proposition 3.16 (Refusal lifting). *$A(\varphi)(\mathcal{P}_1, \dots, \mathcal{P}_n)$ is a protocol on Y . It is nondegenerate at q whenever some $p \in P(q)$ carries a nondegenerate outcome structure.*

Proof. The map obs_q restricted to $V_q^\emptyset \amalg R_q$ is the copairing of maps injective on each summand. Refusal-completeness makes obs_{p_0} injective on $V_{p_0}^\emptyset \amalg R_{p_0}$ and each obs_p injective on R_p . Their images are pairwise disjoint because O_q is the displayed coproduct. A copairing of injections with pairwise disjoint images is injective, so q is refusal-complete. The locus map is the copairing of functions into a common codomain, hence a function into $\mathbf{Nm}$, which is what Theorem 3.11 requires. Nondegeneracy is the observation that R_q contains R_p as a summand for every $p \in P(q)$. $\square$

Write $\mathcal{W}_T^{\text{ts}} = \mathcal{W}_T^{\text{tr}} \cap \mathcal{W}_T^s$ for the strict tree wirings. Let $\mathcal{W}_T^{\text{tso}}$ be its output-supported, positive-arity part. It contains the identities and is closed under positive-arity substitution.

In the statement below, φ and ψ are output-supported strict tree wirings with

$$\varphi \in \mathcal{W}_T^{\text{tso}}(X_1, \dots, X_n; Y), \quad \psi \in \mathcal{W}_T^{\text{tso}}(Z_1, \dots, Z_m; X_i).$$

The port q is the unique output of Y , and p_i is the unique output of X_i .

Lemma 3.17 (Upstream sets compose). *With the notation just fixed,*

$$P_{\varphi \circ_i \psi}(q) = (P_\varphi(q) \setminus \{p_i\}) \amalg \begin{cases} P_\psi(p_i) & \text{if } p_i \in P_\varphi(q), \\ \emptyset & \text{otherwise,} \end{cases}$$

and the displayed union is disjoint.

Proof. Both sides are computed by the recursion (1), which is well founded by acyclicity. If $p_i \notin P_\varphi(q)$ then no walk from q enters X_i , no Z_j is reachable, and the two sides agree. Otherwise, expand (1) for the composite. A composite walk that reaches p_i in φ continues in ψ by Theorem 3.5, and up_ψ returns $\emptyset$ on X_i^{in} , at which point the composite walk resumes in φ . The ports collected in that resumed phase are exactly those of $\text{up}_\varphi(\varphi(d))$ for $d \in X_i^{\text{in}}$, which already lie in $P_\varphi(q) \setminus \{p_i\}$. Disjointness holds because the first summand lies in $\coprod_{i' \neq i} X_{i'}^{\text{out}}$ and the second in $\coprod_j Z_j^{\text{out}}$. The hypothesis that both diagrams are tree wirings is used exactly once. It gives X_i a single output port, so that at most one ψ -expansion occurs. $\square$

Theorem 3.18 (Protocol action on output-supported strict tree wirings). *The assignment $X \mapsto A(X)$ on $\text{Box}_\top^1$, together with the maps $A(\varphi): \prod_i A(X_i) \rightarrow A(Y)$ induced by (2) and (3), is an equivariant, associative, and unital action of $\mathcal{W}_\top^{\text{tso}}$. Equivalently, it is an algebra over the positive-arity suboperad of output-supported strict tree wirings.*

Proof. $A(\varphi)$ is well defined on isomorphism classes because the construction uses only coproducts and copairings, which carry isomorphisms of protocols to isomorphisms of protocols.

Unit. In id_X the single inner box is X itself. For $q \in X^{\text{out}}$ regarded as an outer demand, $\text{id}_X(q)$ is the like-named inner output port q' , and for every inner demand $d \in X^{\text{in}}$, $\text{id}_X(d)$ is the like-named outer input port, on which up is empty by (1). Hence $P(q) = \text{up}(q') = \{q'\}$ and (2) returns $(V_{q'}, V_{q'}^\emptyset, R_{q'}, O_{q'})$ with $\text{obs}_{q'}$, the outcome structure it started from. Equation (3) returns the copairing of the single map $\ell_{q'}$, which is $\ell_{q'}$. So $A(\text{id}_X)$ is the identity on isomorphism classes. This is the step that fails if $P(q)$ is indexed by boxes rather than by ports, since a box with two output ports would then contribute both reason sets at each.

Associativity. By Theorem 3.17, the index set of $A(\varphi \circ_i \psi)$ at q is the disjoint union of $P_\varphi(q) \setminus \{p_i\}$ with $P_\psi(p_i)$. Applying $A(\varphi)$ to $A(\psi)(\mathcal{P}_{Z_1}, \dots, \mathcal{P}_{Z_m})$ in the i -th argument produces $\coprod_{p \in P_\varphi(q)} R'_p$, where $R'_p = R_p$ for $p \neq p_i$ and $R'_{p_i} = \coprod_{r \in P_\psi(p_i)} R_r$. Expanding the p_i summand gives the nested coproduct

$$\left(\coprod_{p \in P_\varphi(q) \setminus \{p_i\}} R_p \right) \amalg \left(\coprod_{r \in P_\psi(p_i)} R_r \right),$$

which is the coproduct over the index set Theorem 3.17 computes for the composite, and the canonical map between them is the flattening bijection β . Both obs and ℓ are copairings out of the coproduct, and β commutes with the injections by construction, so $\text{obs} \circ \beta$ and $\ell \circ \beta$ are the corresponding copairings on the other side. Hence the two protocols are isomorphic and the two isomorphism classes are equal. Locus maps land in the fixed alphabet $\mathbf{Nm}$ and no per-box name set is carried, so nothing belonging to the substituted box X_i appears on either side. Substitution into two distinct inner boxes splits the index set into two summands, to which the argument applies independently.

Equivariance. Let σ be a permutation of $\{1, \dots, n\}$ and $\varphi\sigma$ the reindexed diagram. The construction (2) depends on the inner boxes only through the set $P(q)$ of ports and, for each such port, the protocol on the box that owns it. Reindexing replaces $P_\varphi(q)$ by its image under the induced relabelling of ports, and replaces the tuple $(\mathcal{P}_1, \dots, \mathcal{P}_n)$ by $(\mathcal{P}_{\sigma(1)}, \dots, \mathcal{P}_{\sigma(n)})$. The port-indexed coproducts on the two sides are therefore related by a bijection of index sets under which each summand is the same set. Hence $A(\varphi\sigma)(\mathcal{P}_{\sigma(1)}, \dots, \mathcal{P}_{\sigma(n)}) = A(\varphi)(\mathcal{P}_1, \dots, \mathcal{P}_n)$ as isomorphism classes. $\square$

Remark 3.19 (Why the tree restriction is necessary). Associativity fails on the full operad, and the failure is instructive. Let X_1 have two output ports a and b , let them supply two distinct inputs of a downstream box X_2 , and let the sole output of X_2 supply the output port q of Y . Then

$\{a, b\} \subseteq P_\varphi(q)$. Substitute a diagram ψ into X_1 whose inner boxes include one, say Z_1 , that is upstream of both a and b . Composing in two steps gives R_q a summand R_{Z_1} once for a and once for b . Composing the diagrams first gives it once, because $P_{\varphi \circ \psi}(q)$ is a set. The two composites differ, and the discrepancy is not cosmetic. The two-step reason set reports the same inner refusal twice, which is exactly the over-reporting that Theorem 5.5 shows CatDB refuses at a union node. A treatment of multi-output boxes therefore needs an explicit account of sharing rather than a plain operad. That is the situation of the manifest runtime discussed in Section 6.

Remark 3.20. The mathematics here is elementary: coproducts of injections with disjoint images are injective, and nested coproducts flatten. The engineering content is entirely in the hypothesis that the composite takes O_q to be the coproduct of the inner observation images rather than some quotient of it. That hypothesis is what implementations get wrong. Mapping every inner reason to a string, or to a single **UpstreamError** variant, satisfies a type checker and destroys injectivity. The composite is then a typed interface and not a protocol, and no care at the boundary recovers a distinction discarded inside. Section 5 exhibits an implementation that keeps the summands disjoint by construction, and Section 7 exhibits one that does not.

3.5 Realisation conditions

A *realisation* of the model consists of a type alphabet T , a family $\mathcal{B}$ of boxes over it carrying protocols, a set of diagrams among them, a predicate Valid that an implementation uses to accept diagrams, and an encoding of each port type as a set of transmissible representations. The six conditions below are not all of the same kind, so they are grouped by which of those pieces they constrain.

Definition 3.21 (Conditions on a diagram and its acceptance predicate). Let $\varphi \in \mathcal{W}_{\mathsf{T}}(\vec{X}; Y)$.

- (W1) *Port compatibility.* For every demand port p , $\text{ty}(\varphi(p)) \leq \text{ty}(p)$.
- (W2) *Hereditary checking.* Valid is hereditary: if $\text{Valid}(\varphi)$ then $\text{Valid}(\psi)$ for every sub-diagram ψ of φ . For the recursive validator examined below, this entails a compatibility check at every wire rather than only at the boundary.
- (W3) *Leaf confinement.* There is a distinguished class of *leaf* boxes such that for every φ with $\text{Valid}(\varphi)$ and every $q \in Y^{\text{out}}$,

$$\text{ty}(q) \geq \bigwedge \{ \text{ty}(p) : p \text{ an output port of a leaf of } \varphi \},$$

where the meet is taken in $(\mathsf{T}, \leq)$ and is assumed to exist. No interior box may enlarge the vocabulary beyond what the leaves jointly introduce.

The meet in W3 is not a typographical flourish. In the concrete alphabet $\mathsf{T} = \mathcal{P}_{\text{fin}}(\text{Fld})$ ordered by reverse inclusion, the meet of a family of field sets is their union and $\geq$ is $\subseteq$. So W3 reads: the field set at the root is contained in the union of the field sets introduced at the leaves. Bounding by a single leaf type would be a strictly stronger and false statement for any diagram with more than one leaf, and Theorem 5.2 proves the meet version.

Definition 3.22 (Condition on the protocols). Let $\mathcal{B}$ be the family of boxes of a realisation.

- (W4) *Refusal completeness.* Every box of $\mathcal{B}$ carries a protocol in the sense of Theorem 3.12, and composites are formed by the algebra action of Theorem 3.18, so that inner reason sets enter a composite as summands rather than through a quotient.

The last two conditions are not laws of the operad and are stated separately for that reason. Neither is a property of a diagram φ , and neither can be expressed in the algebra of Theorem 3.18, in which a wire is a syntactic assignment that cannot fail.

Definition 3.23 (Conditions on the deployed family and on port encodings). Let $\mathcal{B}$ be as above and let each port type come with a set of transmissible representations.

- (W5) *Boundary honesty.* The family of boxes a deployment advertises and the family it is willing to answer for are derived from one datum, so the two cannot drift apart. This is a condition on how $\mathcal{B}$ is presented at a deployment, not a condition on any diagram.
- (W6) *Encoding discrimination.* For each port type, the decoding function from transmissible representations is total into $V \amalg R$, and is a coproduct over a closed set of format versions, so an unrecognised version lands in R rather than being coerced into V . Decoding happens at the box that receives the representation, so the resulting reason belongs to that box’s reason set, and the wire itself remains a syntactic assignment.

W1 and W2 are the operad structure made checkable. W3 is a confinement condition with no analogue in the classical wiring-diagram literature, and is the condition with the most content in the system examined below. W4 is Theorem 3.12 restated as an obligation. A realisation satisfying W1 and W2 alone is a typed interface with a compositional type checker. The step to a protocol is W4, and the step to a protocol that can be deployed and versioned is W5 and W6.

3.6 Model faithfulness

Theorem 5.2 below is a theorem about a mathematical model of a Rust enumeration, not about the Rust enumeration. The model is an eight-constructor inductive type with the arities and field payloads of `catdb_algebra::Plan`. Two total functions of and `sc` are defined by the same recursion as the corresponding methods, and a predicate `Valid` by the same recursion as `Plan::validate`. One payload needs care. The field `Aggregate::group_by` has Rust type `Option<String>`. In the model an aggregate node N carries a value $N.g \in \{\text{None}\} \amalg \text{Fld}$, written `Some( $f$ )` when it is the right summand. The `iter` call on it in Listing 9 is `Option::iter`, which yields zero or one element, so the model’s aggregate case yields $\emptyset$ or $\{f\}$. The model is faithful to the extent that three conditions hold. (a) The constructor set is exactly the eight listed, which the implementation enforces by a parallel `NodeKind` enumeration matched exhaustively. (b) The recursions terminate, which holds because the payloads are owned and therefore finite. (c) No other code path constructs a value of the type that bypasses the recursion. Section 7.5 settles (c) for the one construction site outside the crate at this commit and does not claim beyond it. The theorem below is stated for the model, with the implementation function realising each piece named explicitly.

4 The system

Paths in this section and in Section A are written relative to `catdb/crates/` inside the CatDB repository. Two paths are given from the repository root instead and are written out in full: `catdb/proto/openapi.json` and `README.md`.

CatDB at commit `7cc9341a` is a workspace of thirty-four Rust crates that federates external record systems behind one versioned entity model. Callers ask about entities; the system checks the question against a pinned schema, plans what each source can answer, reconciles the results, and returns rows in which each field names the system that answered it. Its README states seven

principles, of which four bear directly on the laws above. Authorization comes first, so policy is evaluated before planning or source disclosure. Provenance is part of the answer, so selected values retain their source record, mapping version and losing candidates. Declarations are hypotheses and never evidence, so a declared relationship cardinality is proven against data before a path answers. And failures stay explicit: refusal, authorization failure, timeout, source failure, a source that was never a candidate, an empty examination and no match are distinct typed outcomes.

4.1 Mode-indexed tool surface

The agent-facing boundary is an MCP server. In the file `catdb-server/src/mcp.rs` the type `CatDbMcp` at line 130 holds an engine handle, a mode and a tool router, and seventeen methods carry a `#[tool]` attribute naming a tool. Two of those names are `catdb.search` and `catdb.entity_query`. Rather than one box with seventeen ports, the faithful reading is a finite family of boxes, one per tool, each with a request port and a refusal-complete response port, of which the server publishes a subset.

```
const LEGACY_TOOL_NAMES: &[&str] = &[
    "catdb.search", "catdb.query", "catdb.read", "catdb.related",
    "catdb.explain", "catdb.validate", "catdb.apply",
    "catdb.build_context", "catdb.index", "catdb.index_status",
];

pub fn published_tools(&self) -> Vec<Tool> {
    let mut tools = self.tool_router.list_all();
    if matches!(self.mode, McpMode::AuthenticatedRemote { .. }) {
        tools.retain(|tool| !LEGACY_TOOL_NAMES.contains(&tool.name.as_ref()));
    }
    tools
}
```

Listing 1: The published surface and the call-time gate are derived from one constant. `catdb-server/src/mcp.rs`, lines 81 and 277, commit 7cc9341a.

The mode is `McpMode`, with variants `EmbeddedInProcess` and `AuthenticatedRemote`. In the embedded mode all seventeen tools are published; in the authenticated remote mode the ten legacy names are removed, leaving seven. The same constant is read by `require_legacy_tool_access` (line 285), which returns an error for exactly those names in the remote mode. That function carries a `debug_assert!` that fires when a tool gates on legacy access without appearing in the list. The stated reason is that a session would otherwise advertise a tool it always refuses. `list_tools` (line 717) and `get_tool` (line 725) both answer from `published_tools`, so discovery and lookup agree.

Proposition 4.1 (Mode-indexed surface). *Let $\mathcal{T}_m$ be the family of tool boxes the implementation publishes in mode m at commit 7cc9341a. Then $\mathcal{T}_{\text{remote}} \subsetneq \mathcal{T}_{\text{embedded}}$, with $|\mathcal{T}_{\text{remote}}| = 7$ and $|\mathcal{T}_{\text{embedded}}| = 17$. Consequently every diagram $\varphi \in \mathcal{W}_{\top}(X_1, \dots, X_n; Y)$ whose inner boxes all lie in $\mathcal{T}_{\text{remote}}$ is also a diagram over $\mathcal{T}_{\text{embedded}}$, and the converse fails for any diagram using one of the ten legacy tools.*

Proof. The counts are the length of `LEGACY_TOOL_NAMES` and the number of `#[tool]` attributes in the file. Inclusion of diagram sets is immediate from inclusion of colour sets: a wiring diagram is a function on demand ports of its inner boxes, and enlarging the ambient family of available boxes does not change any such function. Failure of the converse is the observation that a diagram naming an inner box outside $\mathcal{T}_{\text{remote}}$ is not a diagram over $\mathcal{T}_{\text{remote}}$. $\square$

Theorem 4.1 is small, but it is the kind of statement the wiring reading exists to make. Portability of an agent’s tool composition is monotone in the published surface, and the direction that fails is the one a developer meets. A composition written against a locally embedded CatDB need not survive being pointed at an authenticated deployment. The implementation makes that failure a refusal at publication time rather than at call time.

4.2 The wire format

Write-side traffic is carried by `catdb-ir`. The crate declares a constant `WIRE_VERSION` at line 37, holding the string `"catdb.ir/v1"`, and a type `WireVersion` at line 45. The type is an enumeration with a single variant, which serialises to that same string.

```
pub const WIRE_VERSION: &str = "catdb.ir/v1";

#[derive(Debug, Clone, Copy, PartialEq, Eq,
        Serialize, Deserialize, JsonSchema, Default)]
pub enum WireVersion {
    #[serde(rename = "catdb.ir/v1")]
    #[default]
    V1,
}
```

Listing 2: The wire version is a closed enumeration, not a string. `catdb-ir/src/lib.rs`, lines 37 and 45, commit 7cc9341a.

The payload type is `CatOperation` (line 88) with six variants: `DefineSchema`, `AssertObject`, `RetractObject`, `AssertMorphism`, `RetractMorphism`, `AssertMapping`. Assertion and retraction of a morphism are first-class, which is a categorical vocabulary appearing in the code rather than imposed on it from outside. `OperationEnvelope` (line 144) wraps one operation with the version, an operation identifier used for idempotent application, a workspace and branch identifier, an optional `parent_version` forming an append-only chain, an actor, and a timestamp.

The append-only chain formed by `parent_version` is a wire-level ordering discipline on operations. It is not a state space with a transition structure, and nothing below reads it as one.

4.3 Plan algebra

`catdb-algebra` defines `Plan` (line 71) as an eight-variant enumeration: `Scan` at the leaf, then `Project`, `Filter`, `Join`, `Union`, `Aggregate`, `Sort`, `Limit`, each boxed. A parallel enumeration `NodeKind` (line 385) mirrors the variants and is documented as existing so that adding a ninth node is a compile error in every match rather than a silently skipped branch. The leaf carries its output vocabulary explicitly.

```
pub struct Scan {
    pub target: EntityTarget,
    pub fields: Vec<String>,
    #[serde(default, skip_serializing_if = "Option::is_none")]
    pub source: Option<Uuid>,
}
```

Listing 3: The leaf carries an explicit list of the field names it yields. `catdb-algebra/src/lib.rs`, line 106, commit 7cc9341a.

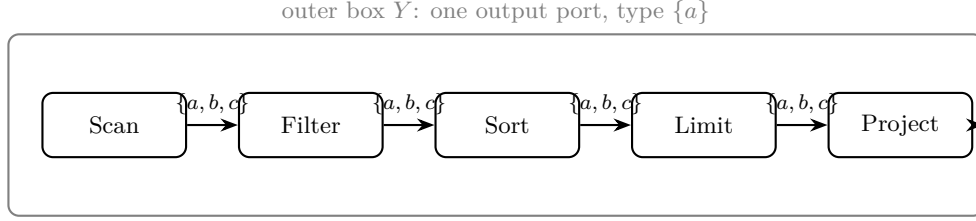


Figure 2: A validated plan read as a tree wiring. The leaf introduces $\{a, b, c\}$; the interior boxes pass it along or narrow it. **Filter** demands b and **Sort** demands c , both supplied. **Project** narrows to $\{a\}$, which is $\leq$ -above the leaf’s type in the reverse inclusion order.

The documentation on `Scan::fields` states that the field list is the scan’s output, listed rather than derived so that field flow through the whole tree can be checked with no catalog present. It states that in federation these are the policy-authorized fields, the exact set `AuthorizedQuery` permits, and that this is why a scan is the only node that can introduce a field name. That is the statement of W3 in the implementation’s own words, and Theorem 5.2 is its proof.

Ownership makes the representation a tree. **Project** owns its input as `Box<Project>` inside the parent variant, **Join** owns a left and a right input, and **Union** owns a vector of inputs. There is no reference type in the enumeration, so a subplan cannot be shared between two parents. This is exactly the linear and rooted condition of Theorem 3.7.

4.4 Typed outcomes and refusals

The refusal surface is not one enumeration but a family, each scoped to a phase of the wiring. The largest is `SourceOutcome` (`catdb-api-types`, line 3347), the outcome of one authorized source execution.

```

pub enum SourceOutcome {
    Ran,                NotApplicable,   Refused,
    Stale,               RuntimeRefused, BudgetExceeded,
    AuthenticationFailed, RateLimited,   IncompleteSearch,
    TransportFailed,    SourceCancelled, SchemaDrift,
    SourceFailed,       ProtocolFailed,  TimedOut,
    SourceTimedOut,     Cancelled,
}

```

Listing 4: Seventeen variants, of which one is success. `catdb-api-types/src/lib.rs`, line 3347, commit 7cc9341a. Doc comments elided.

Two of the doc comments are load-bearing for our reading. `Ran` is documented as covering a successful empty result, which fixes $V^\emptyset$ inside the success summand rather than in the reason set. `NotApplicable` is documented as structurally not a refusal. A refusal means a source that could have been asked declined, whereas this means there was nothing to ask. The comment records that classing it as a refusal previously refused every query in a deployment with more than one entity. That distinction is a statement about the observation map. Two situations that a coarser type would have identified are kept apart because the caller’s next action differs.

The relationship layer carries its own family. In `catdb-core/src/relationship.rs` the type `CardinalityRefusal` at line 946 has five variants, and each one carries the `JoinPath` that was tested.

```

pub enum CardinalityRefusal {
    ContradictedByMeasurement { path: JoinPath, .. },
    DeclaredFanOut             { path: JoinPath, .. },
    Undeclared                 { path: JoinPath, arrow: ArrowName },
    Unmeasured                 { path: JoinPath, .. },
    NothingToExamine           { path: JoinPath, .. },
}

impl CardinalityRefusal {
    pub fn path(&self) -> &JoinPath { /* every variant returns one */ }
}

```

Listing 5: Each refusal names the path it is about. `catdb-core/src/relationship.rs`, lines 946 and 1000, commit 7cc9341a. Payload fields and doc comments elided.

The type documentation states that four refusals exist rather than one because they call for four different fixes: annotate the arrow, rewrite the query, give the probe a source it can run against, or wait for rows to exist. It states that none of them means the join matched nothing, which is instead a successful measurement of zero. The total method `path` is precisely the locus map of Theorem 3.11, realised in code.

Further families follow the same shape. `SourceProbeOutcome` (line 343) has eight variants for the probing phase. `IdentityRefusal` (`catdb-core/src/identity.rs`, line 1469) has documentation recording the deliberate absence of a variant meaning no candidate was offered, on the ground that a caller who never asked is not being refused. `FiberFoldRefusal` (`catdb-query/src/fiber.rs`, line 369) has two variants, each naming the arrow whose declaration is missing or too weak.

4.5 Federation refusal lifting

The federation report shows the composite case. In `catdb-federation/src/report.rs` the type `Outcome` at line 98 has two variants, and the refusing one keeps the inner reason whole.

```

pub enum Outcome {
    Ran(Ran),
    Refused {
        /// The source's own typed refusal, kept whole.
        reason: SourceError,
    },
}

```

Listing 6: The composite keeps the inner reason set rather than flattening it. `catdb-federation/src/report.rs`, line 98, commit 7cc9341a.

This is the coproduct of (2) realised in a type. The composite's reason summand is the inner reason type itself, not a projection of it, so the composite observation map restricts to the inner one. The images of distinct sources stay apart because each `SourceReport` carries its `SourceId`, which is the coproduct tag of Theorem 3.16.

4.6 Provenance and policy

Two crates hold the version-pinned decision records that the correspondence would want to place in Know. `catdb-policy` defines `PolicyEngine` (line 862), a query policy evaluator described in its own doc comment as a pre-planning gate. Alongside it are `Effect` (line 246, variants `Allow` and `Deny`, with `deny` taking precedence), `Action` (line 256, one variant `Query`), and `AuthorizedQuery`

(line 478), an opaque value bound to a catalog revision, a workspace, and a set of authorized entities. `catdb-provenance` defines `FieldObservation` (line 174) and `FieldProvenance` (line 408). The latter carries a provenance rule version, a field name, an optional selected index, and the list of considered observations, with a smart constructor that rejects a selection naming an entry that was not observed.

```
pub fn new(
    provenance_rule_version_id: Uuid, field: impl Into<String>,
    considered: Vec<FieldObservation>, selected: Option<usize>,
) -> Result<Self, ProvenanceError> {
    if let Some(index) = selected {
        let Some(observation) = considered.get(index) else {
            return Err(ProvenanceError::SelectionOutOfBounds {
                selected: index, considered: considered.len(),
            });
        };
        if !observation.state.is_observed() {
            return Err(ProvenanceError::SelectionWasNotObserved { selected: index });
        }
    }
    Ok(Self { provenance_rule_version_id, field: field.into(), selected, considered })
}
```

Listing 7: A provenance record refuses to be built with a selection that names an unobserved candidate. `catdb-provenance/src/lib.rs`, lines 408 and 419, commit 7cc9341a. Abridged.

Both objects have the same shape. Each carries an identifiable statement: this field’s value came from that observation under that rule version, or this principal may query these entities under this catalog revision. Each binds that statement to immutable version identifiers, and each retains enough data for the decision to be re-derived from the same inputs. Neither is a certificate in the sense of Banu’s Definition 1 [1], and neither is called one below. That definition asks for a statement, a parameter assignment, and a derivation that can be mechanically replayed. `AuthorizedQuery` and `FieldProvenance` supply the first two and expose no replayable derivation, and `FieldProvenance::new` is a constructor guard rather than a check that can be re-run against an existing value. Neither is a separately addressable component of the architecture either, and Section 7 takes up what that means.

4.7 Transport surfaces

The file `catdb/proto/openapi.json` is a generated contract with twenty-seven paths under the prefix `/v1/`. Among them are `/v1/query`, `/v1/search`, `/v1/apply`, `/v1/validate`, `/v1/explain`, `/v1/build_context` and `/v1/entity/query`. The MCP tool names and the REST paths correspond one to one on the overlapping subset. The outcome enumerations carry `utoipa::ToSchema` and `schemars::JsonSchema` derivations (`catdb-api-types/src/lib.rs`, lines 3334 to 3344), which is the mechanism by which an OpenAPI document is generated from the same Rust types the MCP surface returns. Whether the checked-in document was regenerated from the types at exactly this commit is not established in general, and that gap is recorded in Section 7.5. It was checked for the two outcome types this paper leans on. The document’s `SourceOutcome` and `SourceProbeOutcome` schemas list exactly the seventeen and the eight variant names, in the same serialised form, that the enumerations declare at 7cc9341a. Subject to the general caveat, the same port types present across two transports without being restated, which is what one wants from a box: the colour is independent of the wire.

Model object	CatDB identifier at 7cc9341a	Where
Type alphabet T , ordered by reverse inclusion	<code>Vec<String></code> field lists on <code>Scan</code> and <code>Project</code>	<code>catdb-algebra/src/lib.rs</code> 106, 165
Box (inner)	one <code>#[tool]</code> method with its request and response types	<code>catdb-server/src/mcp.rs</code> 401 onward
Box (outer), mode-indexed	<code>CatDbMcp</code> with <code>McpMode</code> ; <code>published_tools</code>	<code>catdb-server/src/mcp.rs</code> 130, 277
Wiring diagram, tree fragment	<code>Plan</code> , eight owned variants	<code>catdb-algebra/src/lib.rs</code> 71
Colour discipline on node kinds	<code>NodeKind</code> , closed enumeration matched exhaustively	<code>catdb-algebra/src/lib.rs</code> 385
Supply type of a box	<code>Plan::output_fields</code>	<code>catdb-algebra/src/lib.rs</code> 482
The predicate <code>Valid</code>	<code>Plan::validate</code> , <code>require_fields</code>	<code>catdb-algebra/src/lib.rs</code> 525, 708
Reason set R , execution phase	<code>SourceOutcome</code> , <code>SourceProbeOutcome</code>	<code>catdb-api-types/src/lib.rs</code> 3347, 343
Reason set R , relationship phase	<code>CardinalityRefusal</code> , <code>FiberFoldRefusal</code>	<code>catdb-core/src/relationship.rs</code> 946; <code>catdb-query/src/fiber.rs</code> 369
Locus map ℓ	<code>CardinalityRefusal::path</code> ; <code>arrow</code> fields	<code>catdb-core/src/relationship.rs</code> 1000
Refusal lifting	<code>Outcome::Refused { reason: SourceError }</code>	<code>catdb-federation/src/report.rs</code> 98
Wire version coproduct	<code>WireVersion</code> , <code>WIRE_VERSION</code>	<code>catdb-ir/src/lib.rs</code> 45, 37
Certificate-shaped objects	<code>AuthorizedQuery</code> , <code>FieldProvenance</code>	<code>catdb-policy/src/lib.rs</code> 478; <code>catdb-provenance/src/lib.rs</code> 408

Table 1: Model objects and the identifiers realising them. Every identifier was read at the pinned commit before being listed. Full evidence in Section A.

4.8 Connector conformance

`catdb-sdk-conformance` is a crate whose entire purpose is to be a connector written against the published facade and nothing else; its dependency list names `catdb-sdk` with `default-features = false`, so any gap in the facade is a compile error. Its documentation names four properties a wrong connector gets wrong silently: pushdown honesty, cursor stability, plan dispositions, and uniqueness evidence. Plan dispositions means that every filter a caller asked for is either executed at the source or handed back as a residual, and never dropped. For uniqueness evidence, measured and holds, declared but not enforced, measured and does not hold, and nobody looked are four values rather than one boolean. It also states the three read outcomes it keeps structurally distinct. A read that could not run is an error. A read that examined candidates and matched none is a success with a positive scanned count and no rows. A read with nothing to examine is a success with a zero scanned count and no rows. Those are three values, not one empty list, which is W4 stated for the connector port.

4.9 Mapping table

5 Law checking

5.1 W1, port compatibility

The compatibility check is `require_fields`, a free function at line 708 called from five arms of `Plan::validate`.

```
fn require_fields<'name>(<
  node: NodeKind, input: &Plan,
  referenced: impl Iterator<Item = &'name String>,
) -> Result<(), PlanError> {
  let available = input.output_fields();
  for field in referenced {
    if !available.iter().any(|candidate| candidate == field) {
      return Err(PlanError::FieldNotAvailable {
        node, field: field.clone(), available: available.join(", "),
      });
    }
  }
  Ok(())
}
```

Listing 8: The compatibility check: a node’s referenced fields must be available on its input. `catdb-algebra/src/lib.rs`, line 708, commit 7cc9341a.

In the model of Section 3.6, the type of a supply port is of of the subplan that feeds it and the type of a demand port is the set of names the node references. The condition `require_fields` tests is then $\text{ty}(\varphi(p)) \supseteq \text{ty}(p)$, which is $\text{ty}(\varphi(p)) \leq \text{ty}(p)$ in the reverse inclusion order of Theorem 3.1, so the implementation realises W1 for the field-name alphabet. The error value carries the node kind, the missing field and the available list, which is a locus map and a remedy in one value. The caller learns which operator failed and what it could have asked for.

5.2 W2, hereditary checking

Proposition 5.1 (Hereditary validity). *Let Valid be the predicate defined by the recursion of `Plan::validate` at commit 7cc9341a. Then $\text{Valid}(P)$ implies $\text{Valid}(Q)$ for every subplan Q of P .*

Proof. The body of `Plan::validate` begins by iterating over `self.inputs()` and propagating any error with the question-mark operator, before the match on the node’s own shape is reached. Hence $\text{Valid}(P)$ entails $\text{Valid}(P_i)$ for each immediate input P_i , and the claim follows by induction on the depth of Q in P . The recursion terminates because the payloads are owned and therefore finite. $\square$

The implementation’s own documentation frames this as the property the flat representation could not state. The equivalent checks previously lived as scattered statements inside the query executor, each a one-level check against the request because there was only one level. A sort above a projection that dropped the sort key is a plan the flat form has no way to write, and therefore no way to catch. W2 holds.

5.3 W3, leaf confinement

This is the law with real content. Write $\text{of}(P)$ for the model of `Plan::output_fields` and $\text{sc}(P)$ for the model of `Plan::scans`, the multiset of `Scan` leaves.

```

pub fn output_fields(&self) -> Vec<String> {
  let fields: BTreeSet<String> = match self {
    Self::Scan(scan) => scan.fields.iter().cloned().collect(),
    Self::Project(n)  => n.fields.iter().cloned().collect(),
    Self::Filter(n)   => return n.input.output_fields(),
    Self::Sort(n)     => return n.input.output_fields(),
    Self::Limit(n)    => return n.input.output_fields(),
    Self::Union(n)    => n.inputs.iter().flat_map(|i| i.output_fields()).collect(),
    Self::Join(n)     => n.left.output_fields().into_iter()
                        .chain(n.right.output_fields()).collect(),
    Self::Aggregate(n) => n.group_by.iter().cloned().collect(),
  };
  fields.into_iter().collect()
}

```

Listing 9: The supply type of each node, derived rather than stored. `catdb-algebra/src/lib.rs`, line 482, commit 7cc9341a. Comments elided.

Theorem 5.2 (Field confinement). *Let P be a plan in the model of Section 3.6 satisfying the following two conditions.*

(P) *For every **Project** node N occurring in P , $N.\text{fields} \subseteq \text{of}(N.\text{input})$.*

(A) *For every **Aggregate** node N occurring in P , if $N.g = \text{Some}(f)$ then $f \in \text{of}(N.\text{input})$.*

Then

$$\text{of}(P) \subseteq \bigcup_{s \in \text{sc}(P)} s.\text{fields}.$$

*In particular the conclusion holds whenever $\text{Valid}(P)$. The implementation calls `require_fields` on the **Project** and **Aggregate** arms, and validation is hereditary by Theorem 5.1.*

Proof. Write $L(P) = \bigcup_{s \in \text{sc}(P)} s.\text{fields}$. We induct on the structure of P , noting that sc is defined by the evident recursion, so $L(\text{Filter}(N)) = L(N.\text{input})$ and similarly for **Sort**, **Limit**, **Project** and **Aggregate**, while $L(\text{Union}(N)) = \bigcup_j L(N.\text{inputs}_j)$ and $L(\text{Join}(N)) = L(N.\text{left}) \cup L(N.\text{right})$.

Leaf. $\text{of}(\text{Scan}(s)) = s.\text{fields} = L(\text{Scan}(s))$, so the inclusion holds with equality.

Pass-through nodes. For N of kind **Filter**, **Sort** or **Limit**, Listing 9 returns $\text{of}(N.\text{input})$ directly. By the induction hypothesis this is contained in $L(N.\text{input}) = L(P)$.

Union. $\text{of}(P) = \bigcup_j \text{of}(N.\text{inputs}_j) \subseteq \bigcup_j L(N.\text{inputs}_j) = L(P)$ by the induction hypothesis applied to each input.

Join. $\text{of}(P) = \text{of}(N.\text{left}) \cup \text{of}(N.\text{right}) \subseteq L(N.\text{left}) \cup L(N.\text{right}) = L(P)$, again by the induction hypothesis.

Project. $\text{of}(P) = N.\text{fields}$, which by hypothesis (P) is contained in $\text{of}(N.\text{input})$, which by the induction hypothesis is contained in $L(N.\text{input}) = L(P)$.

Aggregate. If $N.g = \text{None}$ then $\text{of}(P) = \emptyset$ and the inclusion is trivial. If $N.g = \text{Some}(f)$ then $\text{of}(P) = \{f\}$, and $f \in \text{of}(N.\text{input})$ by hypothesis (A), so $\{f\} \subseteq \text{of}(N.\text{input}) \subseteq L(N.\text{input}) = L(P)$ by the induction hypothesis.

All eight constructors are covered, so the induction is complete. $\square$

Two features of the proof are worth naming. Only the **Project** and **Aggregate** cases consume a hypothesis; the other six are closed under the inclusion for structural reasons alone. This is why the theorem holds under (P) and (A) rather than under full validity, and it tells an implementer

which two checks are load-bearing for confinement. The other feature is that the **Union** case relies on `output_fields` taking the union rather than the intersection of its inputs' fields. The code documents that choice on the ground that the intersection would silently drop a field present on every record a branch produced. The looser choice is the one that makes confinement provable, because it never introduces a name. An intersection would be sound here too, but would have made a different property fail instead, completeness of the answer's schema.

Corollary 5.3 (Confinement relative to a leaf constraint). *Let Λ assign a set of field names to each entity target. If every $s \in \text{sc}(P)$ satisfies $s.\text{fields} \subseteq \Lambda(s.\text{target})$, then $\text{of}(P) \subseteq \bigcup_{s \in \text{sc}(P)} \Lambda(s.\text{target})$.*

Proof. Immediate from Theorem 5.2 and monotonicity of union. $\square$

This is a reduction, not a guarantee. It says that a constraint imposed on the leaves propagates to the root, so that checking the leaves suffices. It says nothing about whether the leaves are in fact checked, and it establishes no security property. The intended instance of Λ is the field set an authorization permits for a target, and the next remark records why that instance is not discharged by anything in the crate that establishes the reduction.

Remark 5.4 (Where the hypothesis is not discharged). The crate that establishes the reduction cannot state its hypothesis. `catdb-algebra`'s manifest at commit 7cc9341a lists `catdb-query`, `catdb-api-types`, `serde`, `thiserror` and `uuid` as dependencies, with `catdb-core` only as a development dependency. `catdb-policy` does not appear. `Plan::validate` therefore cannot compare a scan's field list against `AuthorizedQuery`, and does not. The hypothesis of Theorem 5.3 is discharged, if at all, wherever the plan is constructed from an authorization, and nothing examined here relates the two mechanically. The documentation on `Scan::fields` asserts that in federation these are the policy-authorized fields. That is a claim about the construction site, not a check the algebra performs. The accurate summary is that the port discipline moves a field-level obligation from the whole plan to its leaves, and does no more than that. This is the smallest instance of the pattern Section 7 develops. The material one would want to place in `Know` lives in the port type, and the check that the port type is the right one lives somewhere the wiring layer cannot see.

5.4 A linearity condition on unions

Proposition 5.5 (Source linearity). *Let P be a **Union** node with $\text{Valid}(P)$. Then the scan inputs of P that carry a bound source version carry pairwise distinct ones.*

Proof. The **Union** arm of `Plan::validate` inserts each input's `scan.source` into a `BTreeSet` and returns `PlanError::Malformed` when insertion fails, having first skipped inputs that are not scans or that carry no source. Hence a repeated pinned source is rejected. $\square$

The implementation's reason is that the arity of a union node is what reports how many sources carried a record, so a source named twice would inflate the corroboration count while merging the source with itself. In wiring terms this is a linearity condition of a different kind from Theorem 3.7. It does not restrict how many consumers a supply may have. It forbids two inner boxes of the same diagram from denoting the same external system. It has no counterpart in the wiring-diagram literature we know of, because in the classical setting inner boxes are formal and carry no external identity. The scope is narrow: the code compares only inputs that are scans carrying a pinned source, so the condition constrains the pinned fragment and says nothing about unattributed leaves.

5.5 W4, refusal completeness

The evidence is the family of enumerations of Section 4 together with three design decisions visible in the source.

The first decision makes success and empty success one summand and refusals another. `SourceOutcome::Ran` covers a successful empty result by its own documentation. `CardinalityRefusal`'s type documentation states that a join matching nothing is not a refusal but a successful measurement of zero, carried by `CardinalityMeasurement::Observed` with a maximum fan-out of zero. That is $V^\emptyset$ placed inside V and kept out of R .

The second keeps refusals distinct from one another and from adjacent non-refusals. The `NotApplicable` variant exists to hold a case that a coarser type had classed as a refusal, and the doc comment records the consequence of the coarser classification. `IdentityRefusal` records the dual decision, the deliberate absence of a variant for a caller who never asked.

The third makes the transport preserve the distinction. The implementation of `IntoCallToolResult` for `McpApiError` (line 110) routes the typed error body as content text and as a namespaced meta entry rather than as `structuredContent`. The stated reason is that a tool's declared output schema describes its successful payload. A schema-validating client would otherwise replace the reason the server gave with a complaint that the success fields are missing, leaving the caller told the server is malformed rather than that it lacks a grant.

```
fn into_call_tool_result(self) -> Result<CallToolResponse, rmcp::ErrorData> {
    let value = serde_json::to_value(McpApiErrorBody {
        status: self.0.status.as_u16(),
        body: self.0.body,
    })
    .map_err(/* ... */?);
    let mut meta = MetaObject::new();
    meta.0.insert(ERROR_META_KEY.to_string(), value.clone());
    Ok(
        CallToolResult::error(vec![ContentBlock::text(value.to_string())])
            .with_meta(Some(meta))
            .into(),
    )
}
```

Listing 10: The refusal body is routed outside the success schema's jurisdiction. `catdb-server/src/mcp.rs`, line 110, commit 7cc9341a. Abridged.

That is an argument, made in the source, that the observation map must stay injective on refusals across a transport that validates only the success schema. W4 holds on the fragment examined, with the qualification that injectivity of obs was checked by reading the constructors and the transport, not by exhausting all call paths.

5.6 W5, boundary honesty

Listing 1 and Theorem 4.1 give the evidence. Publication and the call-time gate read one constant. `list_tools` and `get_tool` both answer from `published_tools`. A debug assertion fires if a tool gates on legacy access without appearing in the list, with the reason stated in the assertion message. W5 holds for the legacy family in the sense stated, that the advertised and answerable families are derived from one datum.

It does not hold in the stronger sense that every published tool will answer every caller. The seven tools published on an authenticated remote session are subject to per-principal authorization, so a

Condition	Statement	Realising identifier	Status at 7cc9341a
W1	port compatibility	<code>require_fields</code>	Holds for the field-name alphabet; the alphabet carries no base types
W2	hereditary checking	<code>Plan::validate</code>	Holds, Theorem 5.1
W3	leaf confinement	<code>output_fields</code> , <code>Scan::fields</code>	Holds, Theorem 5.2; the authorization hypothesis of Theorem 5.3 is discharged outside the crate
W4	refusal completeness	<code>SourceOutcome</code> , <code>CardinalityRefusal</code> , <code>Outcome::Refused</code>	Holds on the examined fragment, by construction of the outcome types and the transport routing
W5	boundary honesty	<code>LEGACY_TOOL_NAMES</code> , <code>published_tools</code>	Holds for the port set; per-call authorization refusals remain, as intended
W6	encoding discrimination	<code>WireVersion</code>	Not established: unknown versions become serde errors, not typed receiving-box reasons

Table 2: The six conditions against the implementation at the pinned commit. Every status is an implementation-level claim established by source inspection.

published tool can still refuse a particular call. That is the intended behaviour and it is consistent with W4. The refusal is typed, carries a reason, and reaches the caller intact by Listing 10. The condition as stated in Theorem 3.23 is about the published family, not about per-call admissibility. It is not strengthened here, because a protocol whose published ports never refuse would be one with no policy layer at all.

5.7 W6, encoding discrimination

Listing 2 gives the evidence. `WireVersion` is an enumeration with a single variant carrying a serde rename, so deserialising a payload whose version string is anything other than `"catdb.ir/v1"` fails rather than defaulting. The type’s documentation states the intent. Modelling it as an enumeration rather than a bare string means deserialization rejects any unknown version and leaves room to add another variant without changing the enclosing type. This establishes closed version discrimination, but the inspected path returns a serde error rather than a typed member of the receiving box’s reason set. W6 is therefore not established. The MCP and REST request types also carry no version discriminator of this kind at this commit. The REST surface versions itself by path prefix instead, which is a coarser mechanism and is not the same condition.

5.8 Audit summary

6 Layer composition

6.1 Layer contribution

The layer below, `skills`, supplies operations: things that consume typed inputs and produce typed outputs. The layer above, `harness`, supplies the deployment map and the layer at which architecture-level claims are recorded and re-checked, in the sense of [18]. Between them the protocol layer adds a discipline on the connections themselves, and specifically on what happens at a connection when no value can be produced.

That addition is not redundant with either neighbour. A skills operad, as stated in Part II [17], records that an operation has input colours and an output colour. It does not record how many ways the operation can decline, nor whether declining is distinguishable from producing nothing. A harness records checkable statements about an architecture and replays their evidence [1]. A check that fails to replay is a fact about the architecture, not a value that reaches the caller of a wire. Refusal, in the sense of Theorem 3.10, is a value on a wire. It is therefore the protocol layer’s object and not the harness’s.

6.2 Operad distinction

The skills and protocol operads have different colour sets, and neither is a restriction of the other. Both are coloured operads and both call their colours port types, but those colours are different objects and neither is a restriction of the other. $\mathcal{W}$ ’s colours are boxes over $\mathbf{T} = \mathcal{P}_{\text{fin}}(\text{Fld})$ with reverse inclusion, a preorder with genuine content: a supply satisfies a demand when it carries at least the demanded names. $\mathcal{O}$ ’s colours, as stated in Part II [17], are JSON schema expressions together with a top colour for an unconstrained value port and a colour for artifact references, preordered by schema refinement. Its port names are drawn from a countable set of strings keyed to `serde_json::Value` payloads. At the pinned commit almost every port carries the top colour, so the type discipline is trivial almost everywhere.

Proposition 6.1 (Type erasure). *Let e be any assignment sending a *CatDB* port type $S \in \mathcal{P}_{\text{fin}}(\text{Fld})$ to the *AgentHero* port at which a tool response is stored, that is, to a key of *DagIo*. Then e is constant on all port types stored at the same key, so it is not injective as soon as two distinct field sets can be stored at one key. No assignment in the other direction is checked by *DagManifest::validate* at commit `1c3ad24`, because *DagEdge* records only a set of source node names and a set of target node names, and no port datum at all.*

Proof. The first claim is immediate: e is defined by the key at which a response is placed, and a key places no constraint on the JSON value stored there. For the second, *DagEdge* (`crates/dag-runtime/src/lib.rs`, line 636) has two fields, `from` and `to`, each of type `OneOrMany` (line 643), which is one node name or a vector of node names. An edge therefore carries no port type. An edge-local predicate can inspect only its node names, and the current manifest-wide validator performs no producer–consumer schema comparison. The same shape shows that a manifest is not a tree wiring in the sense of Theorem 3.7. An edge with several targets is fan-out, which violates (T1), and a node that writes several named outputs is not a colour of $\mathcal{W}_{\mathbf{T}}^{\text{tr}}$ at all. The validator agrees. *DagManifest::validate* (line 684) checks the concurrency setting, rejects duplicate role and tool identifiers, requires each role’s kind to be accepted by the manifest, and requires a tool’s declared input and output schemas to be mappings. It relates no node’s declared inputs to any producer’s declared outputs. $\square$

```
pub struct DagEdge {
    pub from: OneOrMany,
    pub to: OneOrMany,
}

#[derive(Debug, Clone, PartialEq, Eq, Serialize, Deserialize)]
#[serde(untagged)]
pub enum OneOrMany {
    One(String),
    Many(Vec<String>),
}
```

}

Listing 11: An edge carries names and no port type. `crates/dag-runtime/src/lib.rs` in `AgentHero`, lines 636 and 643, commit `1c3ad24`.

Corollary 6.2. *A composite system in which CatDB tools are wired as nodes of an AgentHero manifest satisfies W1 at the CatDB boundary; at the manifest boundary, W1 is neither enforced nor established, at commits `7cc9341a` and `1c3ad24` respectively.*

The practical reading is that the port discipline is not inherited across the layer boundary. It is enforced inside CatDB, where the plan algebra checks field availability at every node, and it is absent one level up, where the manifest that decides which tool feeds which agent records only names. Any guarantee an agent relies on that is expressed as a port type therefore stops at the database boundary and has to be restated at the manifest level, or not relied upon.

Two further features of the manifest runtime sharpen the contrast, and both are visible in one file. The first concerns arity. Theorem 3.7 requires each box to have one output port, and CatDB’s plan representation satisfies it because a plan node yields one record stream. `DagIo` (`crates/dag-executor/src/lib.rs`, line 76) carries two maps, `values: BTreeMap<String, serde_json::Value>` and `artifacts: BTreeMap<String, ArtifactRef>`, so a node writes an unbounded family of named outputs at once. A box with several output ports is not a colour of $\mathcal{W}_T^{\text{tr}}$. The single-output restriction that costs nothing in a query algebra is therefore a real restriction in a workflow runtime, and it is one reason Theorem 5.2 is a statement about CatDB and not about the layer above.

The second concerns collision. Theorem 5.5 records that CatDB refuses a union whose inputs pin the same source, on the ground that the arity of the node is what reports how many sources carried a record. The manifest runtime resolves the analogous collision instead of refusing it. `DagIo::merge` (line 86) is two calls to `BTreeMap::extend`, which overwrites on a repeated key, so two sibling nodes writing the same output name give a result determined by the order in which they are merged. One system rejects the diagram and the other gives it an order-dependent interpretation, which is the difference between a compositional constraint and none.

6.3 Cross-layer scope

Three statements above are about CatDB and no larger object. Theorem 5.2 is an induction over an owned term, so it applies to the plan representation and not to any wiring with sharing. Theorem 3.18 holds on strict tree wirings and fails on the full operad by Theorem 3.19. Theorem 5.5 constrains scan inputs that carry a pinned source and says nothing about unattributed leaves.

The manifest runtime is outside all three. Its edges relate a set of source names to a set of target names, so fan-out violates (T1) of Theorem 3.7, and its nodes write several named outputs at once, so its boxes are not colours of $\mathcal{W}_T^{\text{tr}}$. A wiring at that level is a general element of $\mathcal{W}_T$, and the results above do not transfer to it. Related work on the harness layer takes up exactly that setting [18].

7 Limits and counterexamples

7.1 Coupled wiring and policy

The correspondence assigns protocols to G and the checkable material to Know. CatDB’s crate graph does not draw that line.

PolicyEngine is evaluated as a pre-planning gate, before the wiring exists. Its own documentation calls it a query policy evaluator and pre-planning gate, and the README states that policy is evaluated before planning, source disclosure, or revision information. So the decision record is not attached to a wiring. It is a precondition on whether a wiring may be built at all. In the other direction, **FieldProvenance** attaches evidence to individual field values inside an executed plan's output. That record is not attached to the wiring either, but to the values travelling on it. Neither is a separately addressable Know component.

We can be more precise than saying the separation fails. **CatDB** puts one fragment of Know into the port types themselves and carries the rest as values in the outcome type.

Remark 7.1 (Checkable material compiled into ports). The authorized field set is the leaf's port label (Listing 3 and the documentation on `Scan::fields`). So the statement that a query touches only authorized fields is not a separate object at all. It follows from the port discipline whenever the leaves respect the authorization, by Theorem 5.3. Checkable statements that cannot be expressed as a port type must go somewhere else, and in **CatDB** they go into the reason sets. A cardinality declaration that the data contradicts becomes `CardinalityRefusal::ContradictedByMeasurement`, carrying the measurement and the tested path. A staleness bound the source exceeded becomes `SourceOutcome::Stale`. A pinned schema identity the provider no longer matches becomes `SourceOutcome::SchemaDrift`. There is no third place. Whatever one would want to call Know in **CatDB** is therefore distributed over T and R , both of which are parts of G on the reading of Theorem 3.12. The absence of a Know crate is not an oversight but the shape this design takes.

This is a genuine cost to the correspondence, and we state it as such. If Know is distributed over the wiring's own type and reason vocabularies, then the triple (G, Know, Φ) is not recovered from **CatDB** by reading off components. It is recovered, if at all, by a modelling choice that decides which part of G to call Know. Part IV reports that its own system separates the two for its hook layer, whose subscriptions can be listed and changed without editing any wiring, and not for its per-operation policy envelopes, which are declared inside the manifest [18]. Neither system separates them completely. The degree of separation is a property of the system rather than of the correspondence, which makes the correspondence's usefulness system-dependent rather than universal.

7.2 Schema crate stub

The pillar mapping invites one to expect the schema layer of a database to carry the port-type semantics. At commit 7cc9341a the file `catdb-schema/src/lib.rs` is eighteen lines long. The whole public interface of the crate is one function, `crate_name`, which returns the string "`catdb-schema`", and the module documentation describes the crate as the later home of a schema language that does not exist yet at this commit.

The real schema material lives in two other crates: `BaseType`, with ten variants, in `catdb-core/src/typeside.rs` at line 12, and `CatOperation::DefineSchema` in `catdb-ir`. The lesson is narrow but worth stating. Crate names in a workspace are a plan, not a decomposition, and a pillar-to-crate mapping read off a directory listing would have assigned this pillar's weight to a file that does nothing. Every mapping in Table 1 names an identifier that was read, for this reason.

7.3 Nominal port labels

The alphabet $T = \mathcal{P}_{\text{fin}}(\text{Fld})$ is thin. `Scan::fields` and `Project::fields` are `Vec<String>`, and no base type travels with them. `catdb-algebra` does not depend on `catdb-core` outside its development dependencies, so `BaseType` is not in scope for the plan type at all. Base types do

enter the plan, but only through predicates. `EntityFilter` (`catdb-query/src/entity.rs`, line 412) carries `attr`, `op`, `value`, `ty: BaseType`, and the schema elements declaring the attribute. `Filter`’s documentation states that it reuses this validated filter rather than defining a second predicate representation, so that the fail-closed matching rules have only one place to drift.

The consequence for W1 is that the compatibility check is name availability, not type agreement. Take a plan in which a `Join` brings two scans together that both expose a field called `id`. It produces an output field set containing one `id`, and nothing in the algebra records which side it came from or whether the two have the same base type. The system answers this at the value level rather than the type level, through per-field provenance (Listing 7), which records for each resolved field the observation selected and the candidates that lost. That is a sound engineering answer and a real weakening of the wiring reading. The diagram does not determine its own semantics at the ports, so the operad structure alone cannot carry the disambiguation. An algebra over $\mathcal{W}_T$ in the sense of Vagner, Spivak and Lerman [12] would need a richer colour set than the one the code uses.

7.4 Tree-fragment implementation

`Plan`’s variants own their inputs, so a subplan cannot be shared. In the vocabulary of Theorem 3.7 the representation maps into $\mathcal{W}_T^{\text{tr}}$, not $\mathcal{W}_T$. Concretely, a query whose optimal shape computes a subresult once and consumes it twice cannot be expressed as a single `Plan`. It must be written as two plans, or the subresult must be recomputed. The wiring reading is therefore accurate about what the code expresses and generous about what the code could express. Any claim that `CatDB` realises Spivak’s operad should be read as mapping plans into the tree-wiring fragment of Theorem 3.9; surjectivity onto that fragment is not claimed.

We do not treat this as a defect. Tree plans are the norm in relational systems for good reasons, and Theorem 5.2 uses the tree structure directly, since the induction is over the structure of an owned term. A shared-subplan representation would need the confinement argument restated over a directed acyclic graph, where the recursion of `output_fields` would need memoisation to remain linear. That is a concrete open problem and it appears in Section 9.

7.5 Threats to validity

The evidence base is source inspection at a single commit, with no execution, so four gaps bound what the audit of Section 5 establishes.

Construction sites. Theorem 5.2 applies to plans satisfying hypotheses (P) and (A), and `Plan::validate` enforces them, so the question is whether a plan can reach the executor unvalidated. At commit 7cc9341a the only crate outside `catdb-algebra` that depends on it is `catdb-federation`. In that crate’s non-test code a `Plan` is constructed at exactly one site, `execute_entity_query` (`catdb-federation/src/service.rs`, line 1764). It builds the plan with `entity_query_plan` at line 2156, calls `validate` at line 2171 and returns the error with the question-mark operator before any source is read, then executes the same value through `execute_row_operators` at line 3313. On that path no unvalidated plan reaches the executor. The remaining gap is narrower than the one stated in earlier versions of this paper. A construction site added later, or one inside `catdb-algebra`’s own test code, is not covered, and the executor function itself does not re-validate its argument.

Reachability of variants. The seventeen `SourceOutcome` variants and the five `CardinalityRefusal` variants are declared. That each is produced on some input was not established, and an unreachable variant contributes nothing to refusal-completeness in practice even though it appears in R .

Injectivity across call paths. The observation map was checked on the constructors and on the MCP transport. A different call path that formats a typed reason into a shared string type would violate refusal-completeness without changing any type declaration examined here.

Currency of the generated contract. `catdb/proto/openapi.json` is a generated artifact checked into the repository. That it agrees with every request and response type at this commit was not established. Two schemas were compared by hand, `SourceOutcome` and `SourceProbeOutcome`, and agree variant for variant with the enumerations at `7cc9341a`. For the remaining types the claim of one surface across two transports rests on the generation procedure rather than on a checked correspondence.

8 Related work

The wiring-diagram operad is due to Spivak [11], with the directed refinement in Rupel and Spivak [10] and the typed-port version, together with the study of algebras over it, in Vagner, Spivak and Lerman [12]. Yau [14] gives a systematic treatment of the coloured operads of wiring diagrams and their algebras, including the finite presentation results we do not need here. Fong and Spivak [4] give the surrounding compositional vocabulary. Our departures are the substitutability preorder on colours, needed because record width subtyping is not equality, and the additional data of Theorem 3.12. We are not aware of a treatment in that line that makes refusal a structure on ports. The closest in spirit is the observation that an algebra over $\mathcal{W}$ assigns behaviours to boxes, since a refusal-complete outcome type is a constraint on which behaviour assignments are admissible.

Within the agent literature, Banu [1] states the correspondence we test, including the integrity labels and optical shapes on wires that we do not find in `CatDB` at this commit. The `ArchAgents` triple is due to de los Riscos, Corbacho and Arbib [9]. Zhou et al. [15] give the four-pillar organisation and Meng et al. [8] the enumerative alternative. Liu’s typed lambda calculus for agent composition [5] is the closest existing formal account of agent-to-agent composition with types. Its reported finding that a large majority of examined compositions are structurally incomplete is consistent with the gap located at the manifest boundary in Theorem 6.2, though the two analyses use different formalisms and we draw no quantitative comparison. Pan et al. [13] treat harnesses as portable, analysable objects, which is the layer above the one studied here. Marom et al. [6] give a category-theoretic framework in an unrelated domain, cited by Banu as cross-domain corroboration that this style of formalisation transfers.

The Model Context Protocol specification [7] is the transport whose tool surface we read as the outer box. It supplies a tool with a name, an input schema and an optional output schema. The absence of any structure on failures beyond an `isError` flag is exactly what makes Listing 10 necessary. `CatDB` routes its typed reason around the specification’s success schema because the specification gives it nowhere else to go. A protocol layer in the sense of Theorem 3.12 would give refusals a schema of their own.

On the database side, the operator-tree representation goes back to Codd [3]. Cao’s argument for why capability is moving out of code [2] is the motivation for treating this interface as a research object at all rather than as an implementation detail.

9 Conclusion

The source audit establishes a useful boundary rather than a system-wide protocol theorem. `CatDB`’s owned plans map into tree wirings, its validator checks field-name availability at each node, and

validated plans satisfy field confinement. The algebraic protocol action additionally requires output-supported strict tree wiring. The manifest runtime above **CatDB** records no port datum and performs no producer–consumer schema comparison, so it neither enforces nor establishes the same compatibility property. Preservation across that boundary remains unproved.

Five problems remain open.

The first is to restate Theorem 5.2 for shared subplans. The confinement argument is an induction over an owned term. A representation admitting common subexpressions replaces the term by a directed acyclic graph. The question is whether the theorem survives with the same two load-bearing hypotheses (P) and (A), and what memoisation the derived supply type needs to stay linear in the size of the graph.

The second is to give the colour set enough structure to make the port-level disambiguation work that Section 7.3 shows is currently done at the value level. Concretely, one wants a colour set of field-name-to-base-type assignments for which the join case of **output_fields** is still definable without a catalog, which is the property the current design bought by using bare names.

The third is to close the erasure gap of Theorem 6.1 from the manifest side. The target is the weakest datum that could be added to a manifest edge such that a validator could reject a wiring whose producer cannot supply what its consumer demands, without requiring the manifest to know the producer’s full response type.

The fourth is to close the gap between the model and the binary. Theorem 5.2 is proved for an inductive model of **Plan**, and Section 3.6 lists the three conditions under which the model tracks the Rust type. The third, that no construction path bypasses the recursion, is established here only for the single construction site outside the crate at this commit. A mechanised statement would need the eight-constructor type, **output_fields** and **validate** transcribed into a proof assistant and the transcription related to the compiled crate, either by extraction or by a Rust verification tool. What the theorem would then buy over the hand proof is not the induction, which is elementary, but the guarantee that the eight cases are the only ones and that the two load-bearing hypotheses are the only ones consumed.

The fifth and most speculative is whether refusal-completeness is preserved by the compiler functors that certificate-preservation arguments concern [1]. Theorem 3.18 shows that refusal-completeness composes within one wiring. It says nothing about translation between architectures, and a translation that merges two reason sets is exactly one that can preserve certificates in Banu’s sense [1] while destroying refusal-completeness.

A Code evidence

Every code-backed claim above has a row below. **CatDB** paths are relative to **catdb/crates/**, except the two written from the repository root; **AgentHero** paths are relative to its repository root. The commits are **7cc9341a** and **1c3ad24** respectively. Every path, identifier and line was read at the stated commit before this table was written. Each entry in the last column is an implementation-level claim from source inspection, not observed runtime behaviour.

Claim	Repository, commit	Path and identifier	Lines	What the code shows
The agent-facing boundary is an MCP tool surface (Section 4)	CatDB 7cc9341a	catdb-server/src/mcp.rs , struct CatDbMcp , tool_router	130, 133	The implementation at 7cc9341a defines an MCP server type holding an engine handle, a mode, and a tool router

Claim	Repository, commit	Path and identifier	Lines	What the code shows
The surface has seventeen tools (Theorem 4.1)	CatDB 7cc9341a	catdb-server/src/mcp.rs, #[tool] attributes	401–689	Seventeen methods carry a <code>#[tool]</code> attribute with a <code>catdb. name</code>
Publication and the call-time gate read one constant (W5)	CatDB 7cc9341a	catdb-server/src/mcp.rs, LEGACY_TOOL_NAMES, published_tools, require_legacy_ tool_access	81, 277, 285	Both functions read the same ten-element constant; a debug assertion fires when a gated tool is absent from it
Discovery and lookup agree (W5)	CatDB 7cc9341a	catdb-server/src/mcp.rs, list_tools, get_tool	717, 725	Both answer from <code>published_tools</code>
Refusal bodies are routed outside the success schema (W4)	CatDB 7cc9341a	catdb-server/src/mcp.rs, IntoCallToolResult for McpApiError	110	The typed error travels as content text and a namespaced meta entry, with <code>structuredContent</code> left absent
Six named tool ports exist with the stated roles (Section 4)	CatDB 7cc9341a	catdb-server/src/mcp.rs, search, query, explain, validate, apply, build_context	401, 417, 465, 481, 497, 513	Each is an async method taking a typed request and returning a typed response or an error
The interior wiring is an eight-variant owned tree (Theorem 3.7)	CatDB 7cc9341a	catdb-algebra/src/lib.rs, enum Plan	71	Eight variants, each payload owned or boxed; no reference type, so no sharing
The node-kind vocabulary is closed	CatDB 7cc9341a	catdb-algebra/src/lib.rs, enum NodeKind	385	A parallel enumeration whose documentation states that a new node forces every match to be revisited
The leaf carries the port label (W3)	CatDB 7cc9341a	catdb-algebra/src/lib.rs, struct Scan, field fields	106	A <code>Vec<String></code> listing the scan’s output field names, documented as the only place a field name enters a plan
The supply type is derived (Theorem 5.2)	CatDB 7cc9341a	catdb-algebra/src/lib.rs, Plan::output_fields	482	An eight-arm match computing the output field set from the node and its inputs
Validity is hereditary (Theorem 5.1)	CatDB 7cc9341a	catdb-algebra/src/lib.rs, Plan::validate	525	The body recurses into <code>self.inputs()</code> and propagates errors before matching the node
Port compatibility is checked pointwise (W1)	CatDB 7cc9341a	catdb-algebra/src/lib.rs, require_fields	708	Returns <code>PlanError::FieldNotAvailable</code> carrying node kind, field, and available list when a referenced name is absent
Unions reject a repeated pinned source (Theorem 5.5)	CatDB 7cc9341a	catdb-algebra/src/lib.rs, Plan::validate, union arm	587–620	Scan inputs carrying a source are inserted into a set and a repeat returns <code>PlanError::Malformed</code>
The algebra cannot see policy (Theorem 5.4)	CatDB 7cc9341a	catdb-algebra/Cargo.toml, [dependencies] and [dev-dependencies]	8–24	Lists <code>catdb-query</code> , <code>catdb-api-types</code> , <code>serde</code> , <code>thiserror</code> and <code>uuid</code> as dependencies at lines 8 to 21, and <code>catdb-core</code> as the sole development dependency at lines 23 and 24; <code>catdb-policy</code> appears in neither

Claim	Repository, commit	Path and identifier	Lines	What the code shows
Base types travel with predicates, while port types encode bare field names (Section 7.3)	CatDB 7cc9341a	catdb-query/src/entity.rs, struct EntityFilter, field ty; catdb-algebra/src/lib.rs, Scan::fields and Project::fields	412; 106, 165	EntityFilter declares ty::BaseType alongside the declaring schema origins, whereas the two fields carrying a plan's field-name port types are declared Vec<String> with no base-type component
A base-type vocabulary exists elsewhere	CatDB 7cc9341a	catdb-core/src/typeside.rs, enum BaseType	12	Ten variants including Vector(u16) and TextArray
Success, including empty success, is one variant and the ways of declining are separate variants (W4)	CatDB 7cc9341a	catdb-api-types/src/lib.rs, enum SourceOutcome	3347	Seventeen variants of which Ran is the only success; its documentation states that it covers a successful empty result, so empty success sits inside the success variant rather than among the other sixteen, and NotApplicable is documented as structurally not a refusal
A second reason set is scoped to probing	CatDB 7cc9341a	catdb-api-types/src/lib.rs, enum SourceProbeOutcome	343	Eight variants covering verified, drifted, unauthorized, unavailable, timeout, cancelled, rate limited, unsupported
Refusals carry a locus (Theorem 3.11)	CatDB 7cc9341a	catdb-core/src/relationship.rs, enum CardinalityRefusal, CardinalityRefusal::path	946, 1000	Five variants each carrying a JoinPath ; path is total and returns one for every variant
A refusal set may deliberately omit a variant	CatDB 7cc9341a	catdb-core/src/identity.rs, enum IdentityRefusal	1469	Documentation states there is deliberately no variant meaning no candidate was offered
Refusal reaches the query layer	CatDB 7cc9341a	catdb-query/src/fiber.rs, enum FiberFoldRefusal	369	Two variants, each naming the arrow whose declaration is missing or admits shared members
Composites keep inner reasons whole (Theorem 3.16)	CatDB 7cc9341a	catdb-federation/src/report.rs, enum Outcome	98	The refusing variant carries the source's own typed SourceError rather than a projection
The wire format is a closed version coproduct (W6)	CatDB 7cc9341a	catdb-ir/src/lib.rs, WIRE_VERSION, enum WireVersion	37, 45	One variant with a serde rename to "catdb.ir/v1", so an unknown version fails to deserialize
The write vocabulary names morphisms	CatDB 7cc9341a	catdb-ir/src/lib.rs, enum CatOperation, struct OperationEnvelope	88, 144	Six operations including AssertMorphism and RetractMorphism ; the envelope carries version, operation id, workspace, branch, parent version, actor, timestamp

Claim	Repository, commit	Path and identifier	Lines	What the code shows
Policy is a pre-planning gate (Section 7.1)	CatDB 7cc9341a	catdb-policy/src/lib.rs, struct PolicyEngine, enum Effect, enum Action, struct AuthorizedQuery	862, 246, 256, 478	The type's documentation calls it a query policy evaluator and pre-planning gate; deny takes precedence and the authorization is bound to a catalog revision
Provenance is recorded per field of a source record (Theorem 7.1)	CatDB 7cc9341a	catdb-provenance/src/lib.rs, struct FieldObservation, struct FieldProvenance, FieldProvenance::new	174, 408, 419	Each record names one field and carries a source version, a mapping version, a record key and the considered candidates; the constructor returns SelectionWasNotObserved when the selected index names a candidate whose state is not observed
The connector contract keeps three read outcomes distinct (W4)	CatDB 7cc9341a	catdb-sdk-conformance/src/lib.rs, crate documentation, MinimalSource	1–66, 100	The documentation states that a read that could not run, a read that matched none, and a read with nothing to examine are three values, and names four properties a wrong connector loses silently
The generated REST contract declares twenty-seven versioned paths	CatDB 7cc9341a	catdb/proto/openapi.json, paths under /v1/	whole file	Twenty-seven paths, including /v1/query, /v1/explain, /v1/validate and /v1/entity/query, whose names correspond to MCP tool names on the overlapping subset
The response types carry OpenAPI schema derivations	CatDB 7cc9341a	catdb-api-types/src/lib.rs, derive list on enum SourceOutcome	3334–3344	The enumeration derives utoipa::ToSchema and schemars::JsonSchema alongside Serialize and Deserialize
The one non-test construction site outside the algebra validates before executing (Section 7.5)	CatDB 7cc9341a	catdb-federation/src/service.rs, execute_entity_query, entity_query_plan, Plan::validate, execute_row_operators	1764, 2156, 2171, 3313	The function builds the plan, calls validate and propagates its error before reading a source, and later passes the same plan to the row-operator executor; the crate is the only one outside catdb-algebra that depends on it, and its other two validate calls lie in test modules
The checked-in contract agrees with the two outcome enumerations (Section 7.5)	CatDB 7cc9341a	catdb/proto/openapi.json, components.schemas. SourceOutcome, SourceProbeOutcome	whole file	The SourceOutcome schema lists seventeen snake-case names and the SourceProbeOutcome schema eight, equal as sets and in order to the variants of the two enumerations at this commit

Claim	Repository, commit	Path and identifier	Lines	What the code shows
The schema crate is a stub (Section 7.2)	CatDB 7cc9341a	catdb-schema/src/lib.rs, fn crate_name	1–18	The crate’s whole public interface is one function returning its own package name
The design principles are stated in prose	CatDB 7cc9341a	README.md, principles list	22–41	States that authorization comes first, provenance is part of the answer, declarations are hypotheses, and a failure never looks like an empty result
A manifest edge carries no port type (Theorem 6.1)	AgentHero 1c3ad24	crates/dag-runtime/src/lib.rs, struct DagEdge, enum OneOrMany	636, 643	Two fields, each a string or a vector of strings; no type datum on an edge
The manifest validator checks shape, not port agreement (Theorem 6.2)	AgentHero 1c3ad24	crates/dag-runtime/src/lib.rs, struct DagManifest, DagManifest::validate	658, 684	Checks concurrency, duplicate roles and tools, accepted kinds, and that tool schemas are mappings; no check relates a node’s inputs to a producer’s outputs

References

- [1] B. Banu. Harness engineering as categorical architecture: structural guarantees are harness-level properties. arXiv:2605.12239, 2026.
- [2] Z. Cao. The end of software engineering: how AI agents are fundamentally restructuring the software paradigm. arXiv:2606.05608v1, 2026. Retitled “Agentic software: how AI agents are restructuring the software paradigm” in the current version of the record.
- [3] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970. DOI 10.1145/362384.362685.
- [4] B. Fong and D. I. Spivak. Seven sketches in compositionality: an invitation to applied category theory. arXiv:1803.05316, 2018.
- [5] Q. Liu. λ_A : a typed lambda calculus for LLM agent composition. arXiv:2604.11767, 2026.
- [6] L. Marom, S. Tibbits, G. Zardini, and M. J. Buehler. A category-theoretic framework from biological mechanics to engineered stimulus-response systems. arXiv:2604.26367, 2026.
- [7] Model Context Protocol specification, revision 2025-06-18. <https://modelcontextprotocol.io/specification/2025-06-18>. Accessed September 2026.
- [8] Q. Meng, Y. Wang, L. Chen, W. Wu, Y. Li, W. Jiang, Q. Wang, C. Lu, Y. Gao, Y. Wu, and Y. Hu. Agent harness for large language model agents: a survey. Preprints 202604.0428, version 2, 2026. DOI 10.20944/preprints202604.0428.v2.
- [9] P. de los Riscos, F. J. Corbacho, and M. A. Arbib. Working paper: towards a category-theoretic comparative framework for artificial general intelligence. arXiv:2603.28906, 2026.
- [10] D. Rupel and D. I. Spivak. The operad of temporal wiring diagrams: formalizing a graphical language for discrete-time processes. arXiv:1307.6894, 2013.

- [11] D. I. Spivak. The operad of wiring diagrams: formalizing a graphical language for databases, recursion, and plug-and-play circuits. arXiv:1305.0297, 2013.
- [12] 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.
- [13] L. Pan et al. Natural-language agent harnesses. arXiv:2603.25723, 2026.
- [14] D. Yau. *Operads of Wiring Diagrams*. Lecture Notes in Mathematics 2192, Springer, 2018. arXiv:1512.01602.
- [15] 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.
- [16] M. Long. Coalgebraic Memory. Part I of the Agentic Engineering series, The YonedaAI Collaboration, YonedaAI Research Collective, 2026.
- [17] M. Long. Operadic Skill Composition. Part II of the Agentic Engineering series, The YonedaAI Collaboration, YonedaAI Research Collective, 2026.
- [18] M. Long. Harness Architecture. Part IV of the Agentic Engineering series, The YonedaAI Collaboration, YonedaAI Research Collective, 2026.