

Operadic Skill Composition

Part II of *Agentic Engineering*

Matthew Long

The YonedaAI Collaboration, YonedaAI Research Collective

Chicago, IL

`matthew@yonedaai.com`

September 2026

Abstract

The claim that skills are operations of a coloured operad is testable only after its colours, operations, substitution, and semantics are specified. We define such an operad over port types and compare it with a coloured PROP, which preserves multi-output nodes as single operations. Seven conditions are checked against the directed-acyclic-graph runtime of **AgentHero** at commit **1c3ad24**. Validation establishes a finite acyclic dependency order, scheduling is invariant under irrelevant presentation brackets, and concurrently scheduled nodes pass an interference check. Validation does not establish input closure against an initial-store interface, the readable port set can depend on the surrounding composite, and the manifest language has no generic identity node. Actual disjointness of sibling outputs is sufficient for order-independent merging; a collision with distinct values is a counterexample, while declared disjointness alone is insufficient because value outputs are not confined to declarations. A **Set**-valued algebra can represent deterministic correlations among several outputs but not nondeterministic results at a fixed input. For manifests whose layers are all scheduled concurrently, handler-entry orders define one Mazurkiewicz trace class. The evidence is source inspection at a pinned commit, not execution or machine-checked proof.

Contents

1	Introduction	3
1.1	Problem	3
1.2	Scope	3
1.3	Contributions	4
1.4	Series context	4
1.5	Evidence method	4
2	Background	5
2.1	The Skills pillar	5
2.2	Coloured operads	5
2.3	Wiring diagrams	7
2.4	Audit criteria	7
3	The formal model	7
3.1	Ports, colours and interfaces	7

3.2	Skills as operations	8
3.3	Operads and multi-output structure	8
3.4	Composites and manifests	9
3.5	Effect grading	10
3.6	The laws	10
4	The system	11
4.1	Runtime	11
4.2	The mapping	11
4.3	The operation signature	11
4.4	The kind vocabulary	13
4.5	A worked manifest	13
4.6	Schema discipline	14
5	Checking the laws	14
5.1	Law L7: finite dependency order	14
5.2	Law L3: schedule invariance	15
5.3	Law L6: checked independence	16
5.4	Law L1: input closure	17
5.5	Law L5: composite-dependent reads	18
5.6	Law L4: equivariance	19
5.7	Law L2: manifest identity representation	20
5.8	Summary	20
6	Derived operations	21
6.1	Gate and approval as guards	21
6.2	Branch as an indexed family	21
6.3	Map as bounded fan-out	22
6.4	Bounded loops	22
6.5	Sub-DAG invocation	23
6.6	Enforced colour discipline	23
7	Multi-output nodes	24
7.1	Multi-output structure	24
8	Layer composition	25
8.1	Layer contribution	25
8.2	Lower-layer composition	25
8.3	Upper-layer composition	26
8.4	The policy envelope	26
8.5	Layer dependencies	26
9	Limits and counterexamples	26
9.1	Open colour set	26
9.2	Multi-output nodes	27
9.3	Post-validation coordination	27
9.4	Effectful operations	27
9.5	Unenforced colour declarations	28
9.6	Threats to the reading	28

10 Related work	28
11 Conclusion	29
12 Open problems	29
A Code evidence	30

1 Introduction

1.1 Problem

An agent harness is the machinery around a language model: the prompts, the tool surface, the memory, the scheduler. Zhou et al. [27] organise this machinery into four externalization pillars, Memory, Skills, Protocols and Harness. Banu [2] proposes that the pillars already have a formal theory, the ArchAgents Architecture triple (G, Know, Φ) of de los Riscos, Corbacho and Arbib [4]. In that correspondence the Skills pillar is assigned a categorical role: skills are objects composed via an operad, with three named composition operations, serial, parallel and trace.

The word “operad” carries content. A coloured operad is not a synonym for a graph of tasks. It is a structure with a fixed colour set, operations typed by input and output colours, a substitution operation, a symmetric group action, and three axioms that these must satisfy [9, 5, 26]. A runtime called an operad owes an account of its colours, its operations, what substitution means, and whether the axioms hold. Otherwise the claim is a metaphor, and the failure modes of the runtime are exactly the places the metaphor hides.

The correspondence names a coloured operad whose colours are port types and whose operations are skills, with serial, parallel, and trace composition [2]. That choice is the hypothesis under test. Because an operad has one output per operation while runtime nodes may have several, a coloured PROP is the necessary comparison structure. Validator and scheduler results apply to both; output representation and interchange do not.

This paper does that work for one system. **AgentHero** is a production orchestration platform whose unit of work is a validated DAG manifest: a document declaring nodes, edges, roles, tools and policies, executed by a generic executor parameterised over an application supplied handler. The manifest looks operadic. Nodes have declared inputs and outputs. Edges connect nodes. There are combinators named **DagLoop**, **DagMap**, **DagBranch** and **DagGate** that resemble trace, parallel, conditional and guard. The question is whether the resemblance survives contact with the code.

1.2 Scope

We do not claim that the Skills pillar and a coloured operad are the same thing. The claim is narrower. Under stated assumptions there is a structure-preserving interpretation of the Skills pillar as the operations of a coloured operad whose colours are port types. One production system realises that interpretation on an identifiable fragment, and this paper says where the interpretation fails. The fragment turns out to be small and precisely describable, and the stated assumptions turn out to be load-bearing. The main assumption is that a countable, open, string-keyed set of port names can stand in for a colour alphabet. It is stated as Theorem 3.10, and Section 9 examines what it costs.

1.3 Contributions

The paper defines the skills operad $\mathcal{O}$, a corresponding PROP, and seven source-decidable conditions on a runtime realisation. Applied to **AgentHero** at commit `1c3ad24`, the audit separates properties of dependency order and scheduling from properties the manifest validator does not establish. The complete verdict appears in Table 3.

Two results locate the semantic boundary. Concurrent layers determine a Mazurkiewicz trace class (Theorem 5.8), while a last-write-wins collision prevents execution from factoring through $\mathcal{O}$ (Theorem 5.18). A PROP preserves multi-output invocation identity, but interchange still requires actual output disjointness and strand-local reads (Theorem 7.4). Every implementation claim is tied to a pinned identifier in Section A; none reports live execution.

1.4 Series context

This paper is self-contained: every definition it uses is stated here, and every result it proves is proved here. Three companion papers treat the other pillars. Their subject matter overlaps at the boundaries, so we say exactly where, and no object is defined twice under two names.

The port-type alphabet $\mathbf{T}$ is developed in the companion paper on protocols [11]. That paper also settles the relation between the wiring operad $\mathcal{W}$ studied there and the skills operad $\mathcal{O}$ studied here. They are distinct operads with different concrete colour sets, related in one direction by a type erasure with no manifest-level section. We use $\mathbf{T}$ only as a generic set of port types and define this paper’s concrete colours below. The Architecture triple (G, Know, Φ) and the certificate layer **Know** are developed in the companion paper on the harness [12]. This paper defines no notion of certificate, and where it observes that a code object is certificate-shaped it says so and stops. The memory carrier of [10] enters here only as one colour that a port may carry, written **Mem**; the coalgebra itself is neither restated nor used. None of these imports is load-bearing for a proof. The only property of $\mathbf{T}$ used below is that it is a set of colours, and the concrete alphabet analysed here is defined in full in Theorem 3.2. No result here mentions a certificate, and **Mem** occurs only as a name. A reader without the companion papers loses context and no definition.

Two dependencies run against the presentation order **Memory**, **Skills**, **Protocols**, **Harness**, and are noted because a reader might otherwise take the order for a dependency order. The port-type alphabet comes from the protocols paper, which is presented after this one. A skill may read and write memory, so the memory carrier can inhabit a colour of $\mathcal{O}$, while nothing in the memory paper depends on anything here. Neither is a circular definition.

1.5 Evidence method

Every claim about code in this paper is a claim about the state of a named file at a named commit, established by reading that file. The pinned commits are **AgentHero** `1c3ad24011081cd1c15f629ae5fd635237d595b3` (short `1c3ad24`) and, as a secondary system, **agent-os** `61cb3994daf66eae18322bd01853df68d245a681` (short `61cb399`). Statements take the form “the implementation at commit `1c3ad24` performs Y ”. No claim in this paper is a report of a measured run, a benchmark, or observed production behaviour, and none should be read as one. Section A lists every code-backed claim with its path, identifier, line range and commit.

Two kinds of statement appear below and they are kept apart. A result labelled Theorem, Lemma, Proposition or Corollary is a mathematical statement about the model of Section 3, proved from the definitions. A result labelled code observation is a statement about the text of a named source file at a named commit; its justification is headed “Source inspection” and consists of exhibiting the relevant lines. A code observation is not a proof about the behaviour of the compiled program: it rests on

no formal semantics of Rust, and nothing here is machine checked. Where a mathematical result depends on a code observation, that observation appears as an explicit premise in the statement, so the implication and the empirical premise can be assessed separately. Such a result asserts the implication and nothing more. The claim that its antecedent holds of the source is made only by the code observation itself, at the evidential standard of Section A. A reader who rejects the observation rejects the conclusion without any mathematics having failed.

Availability of the evidence. **AgentHero** is a private production repository, so a reader cannot clone it. Two things are done in compensation, and neither is a substitute for an openly archived artifact. Every claim carries an exact path, identifier and line range at a fixed commit, so it is checkable by anyone with repository access and falsifiable by them. The source that carries the load-bearing claims is quoted in the listings, with any elision stated in the caption. The reasoning from the source text to the claim can therefore be checked from this document alone. Section 9 records the limitation.

Two naming hazards are worth stating at the outset. The identifiers **SkillStage**, **PatternTemplate**, **WiringDiagram**, **SkillOrganism**, **RunContext** and **BiTemporalMemory** belong to Banu’s own reference implementation [2] and do not occur in **AgentHero** at 1c3ad24. **AgentHero** independently realises an analogous structure under different names; it does not implement that vocabulary. The secondary repository **agent-os** contains a paper of its own titled “The AI Operating System, Part II: Tool Interface Layer”. That Part II is not this Part II. Where we cite **agent-os** we cite it as a prior, same-author case study in a different language and runtime.

2 Background

2.1 The Skills pillar

Zhou et al. [27] describe externalization as the movement of agent capability out of the model’s weights and into inspectable, reusable artifacts. Skills are the pillar that holds procedural capability: a named unit of work with a described interface, invoked by an agent rather than reconstructed by it each time. The industrial form of this idea is a manifest format for a capability, as in the Agent Skills standard [1] and the Model Context Protocol tool surface [19].

One result shapes the pillar in a way this paper can use. Liu [14] gives a typed lambda calculus for agent composition, with mechanised proofs, and reports that a large majority of the compositions found in five deployed frameworks are structurally incomplete with respect to that calculus. That is a statement about the discipline of composition rather than about any one system, and it is the gap a compositional formalism is meant to close. The correspondence this paper tests also appeals to an empirical claim that atomic coding skills compose without negative interference, citing [15]. That preprint has since been withdrawn by its authors. No argument here rests on it, and it is cited only to identify what the correspondence appeals to.

Banu [2] closes it by assignment: Skills are “objects composed via operad”, with serial composition $B \circ A$, parallel composition $A \otimes B$, and trace $\text{Tr}(A)$. The assignment is stated at the level of a correspondence table and is not accompanied by a definition of the operad’s colours or a check of the operad axioms in the reference implementation. That is the gap this Part addresses, on a different system.

2.2 Coloured operads

We recall the definition we will use. Nothing here is new. See Leinster [9] for the general theory, Fong and Spivak [5] for an introduction aimed at applied categorists, and Yau [26] for the wiring-diagram

case.

Definition 2.1 (Coloured symmetric operad). Let $\mathbf{C}$ be a set, whose elements are called colours. A $\mathbf{C}$ -coloured symmetric operad $\mathcal{O}$ consists of

1. for each $n \geq 0$, each tuple $(c_1, \dots, c_n) \in \mathbf{C}^n$ and each $c \in \mathbf{C}$, a set $\mathcal{O}(c_1, \dots, c_n; c)$, whose elements are called operations of arity n ;
2. for each $c \in \mathbf{C}$ an identity $\text{id}_c \in \mathcal{O}(c; c)$;
3. composition maps

$$\gamma: \mathcal{O}(c_1, \dots, c_n; c) \times \prod_{i=1}^n \mathcal{O}(\vec{d}_i; c_i) \longrightarrow \mathcal{O}(\vec{d}_1, \dots, \vec{d}_n; c);$$

4. a right action of the symmetric group Σ_n sending $f \in \mathcal{O}(c_1, \dots, c_n; c)$ and $\sigma \in \Sigma_n$ to $f \cdot \sigma \in \mathcal{O}(c_{\sigma(1)}, \dots, c_{\sigma(n)}; c)$;

subject to associativity of γ , the unit laws $\gamma(f; \text{id}_{c_1}, \dots, \text{id}_{c_n}) = f = \gamma(\text{id}_c; f)$, and equivariance: γ commutes with the symmetric actions on the outer operation and on the list of inner operations.

Definition 2.2 (Partial composition). For $f \in \mathcal{O}(c_1, \dots, c_n; c)$, $1 \leq i \leq n$ and $g \in \mathcal{O}(\vec{d}; c_i)$, partial composition is

$$f \circ_i g = \gamma(f; \text{id}_{c_1}, \dots, \text{id}_{c_{i-1}}, g, \text{id}_{c_{i+1}}, \dots, \text{id}_{c_n}) \in \mathcal{O}(c_1, \dots, c_{i-1}, \vec{d}, c_{i+1}, \dots, c_n; c).$$

The two forms determine each other; the axioms of Theorem 2.1 are equivalent to the two partial-composition associativity identities together with equivariance and the unit laws.

Partial composition is the substitution of one skill into one input slot of another. In diagrammatic form, for $g \in \mathcal{O}(\vec{d}; c_i)$ the assignment $f \mapsto f \circ_i g$ is a map of hom-sets, and the two ways of substituting into two distinct slots agree:

$$\begin{array}{ccc} \mathcal{O}(c_1, \dots, c_n; c) & \xrightarrow{-\circ_i g} & \mathcal{O}(\dots, \vec{d}, \dots; c) \\ \downarrow -\circ_j h & & \downarrow -\circ_{j'} h \\ \mathcal{O}(\dots, \vec{e}, \dots; c) & \xrightarrow{-\circ_i g} & \mathcal{O}(\dots, \vec{d}, \dots, \vec{e}, \dots; c) \end{array}$$

Here $i < j$, and $j' = j + |\vec{d}| - 1$ is the index of the j -th slot after the first substitution has widened the list. Commutativity of this square is the parallel case of associativity: for $i < j$, $g \in \mathcal{O}(\vec{d}; c_i)$ and $h \in \mathcal{O}(\vec{e}; c_j)$,

$$(f \circ_i g) \circ_{j+|\vec{d}|-1} h = (f \circ_j h) \circ_i g. \quad (1)$$

Informally: plugging two independent skills into two distinct input slots of a third gives the same composite whichever one is plugged in first. Equation (1) is the axiom that a scheduler with an order-sensitive output store can violate, and Theorem 5.16 says exactly when **AgentHero** violates it.

Definition 2.3 (Free operad on a signature). A $\mathbf{C}$ -coloured signature Σ assigns to each $(c_1, \dots, c_n; c)$ a set $\Sigma(c_1, \dots, c_n; c)$ of generators. The free $\mathbf{C}$ -coloured symmetric operad $\text{Fr}(\Sigma)$ has as its operations the isomorphism classes of finite planar rooted trees, modulo the symmetric action on leaf orderings. Internal vertices of such a tree are labelled by generators of matching colours and its leaves are labelled by colours. Composition is grafting.

Free operads matter here because a manifest is a syntactic object. It declares a composite; it does not by itself say what the composite computes. The distinction between the free operad of composites and an algebra over it is where most of the honest content of Section 5 lives.

Definition 2.4 (Algebra over an operad). An algebra for $\mathcal{O}$ in **Set** assigns to each colour c a set A_c and to each operation $f \in \mathcal{O}(c_1, \dots, c_n; c)$ a function $A_f: A_{c_1} \times \dots \times A_{c_n} \rightarrow A_c$, compatibly with γ , the units and the symmetric action.

2.3 Wiring diagrams

Spivak’s operad of wiring diagrams [21], its directed refinement [20] and the algebra of open dynamical systems over it [22] are the standard formalisation of “boxes wired together”. They are the natural home for the Protocols pillar, and Part III [11] develops the wiring operad $\mathcal{W}$ there. This Part uses the operad formalism, not the wiring-diagram operad specifically. As stated in Part III, $\mathcal{W}$ and $\mathcal{O}$ are distinct operads with different concrete colour sets. The only relation between them that is checkable at the pinned commits is a type erasure in one direction. **AgentHero**’s **DagEdge** records node identifiers only, so the erasure has no manifest-level section. We cite that statement and do not restate it.

2.4 Audit criteria

An operadic account must specify its colours, operations, substitution, axioms, and algebra. The model below makes each obligation explicit enough to test against the runtime.

3 The formal model

3.1 Ports, colours and interfaces

Definition 3.1 (Port name). Fix a countably infinite set **Name** of port names. In **AgentHero**, **Name** is the set of strings admissible as keys of **DagIo**, a structure with two named maps, one from strings to JSON values and one from strings to artifact references.

In the sense of [11], a typed port is a pair (k, τ) with $k \in \mathbf{Name}$ and τ drawn from a port-type alphabet $\mathbf{T}$. We now say what alphabet **AgentHero** supplies.

Definition 3.2 (The colour set). Let **Sch** be the set of schema expressions accepted by the manifest schema checker. These are finite trees built from the keywords `type`, `enum`, `minimum`, `maximum`, `minItems`, `maxItems`, `minLength`, `maxLength`, `required`, `properties`, `items` and `additionalProperties`, where `type` ranges over `object`, `array`, `string`, `number`, `integer`, `boolean` and `null`. The **AgentHero** colour set is

$$\mathbf{C}_{\text{AH}} = \{\top\} \cup \{\text{json}[\sigma] : \sigma \in \mathbf{Sch}\} \cup \{\text{artifact}\},$$

preordered by $\text{json}[\sigma] \leq \top$ for every σ , and $\text{json}[\sigma] \leq \text{json}[\sigma']$ when every value satisfying σ satisfies σ' . Refinement is a preorder and not a partial order, since distinct schema expressions can accept the same values; $\mathbf{C}_{\text{AH}}$ denotes the quotient by the induced equivalence, so that $\leq$ is a partial order on it. Nothing below depends on the choice of representative. Here $\top$ is the top element, the colour of a value port carrying an unconstrained JSON value; $\text{json}[\sigma]$ is the colour of a value port whose values satisfy σ ; and **artifact** is the colour of an artifact-valued port.

Remark 3.3. $\mathbf{C}_{\text{AH}}$ has a top element, and Theorem 5.10 shows that at commit 1c3ad24 almost every port carries it. A colour lattice whose typical element is the top element imposes no discipline. This is the technically exact form of the objection that **AgentHero**’s ports are “dynamically typed”: the type system exists, it is just almost everywhere trivial.

Definition 3.4 (Interface). An interface is a finite partial function $u: \mathbf{Name} \rightarrow \mathbf{C}$. Write $\text{dom}u$ for its domain. Interfaces are ordered by the ambient order on names: since **AgentHero** stores port maps in key-sorted order, every interface has a canonical enumeration $k_1 < \dots < k_n$ of $\text{dom}u$.

Remark 3.5. Theorem 3.4 removes one apparent difficulty. A coloured operad’s operations have *ordered* input tuples, whereas **AgentHero**’s node inputs are *named*. The canonical key order supplies the missing order, and it does so uniformly, so the symmetric group action on the input side is carried by relabelling and is not itself a source of ambiguity. The ambiguity that does arise appears on the output side, in Theorem 5.16.

3.2 Skills as operations

Definition 3.6 (Operation descriptor). An operation descriptor is a quadruple $N = (q, u, v, \kappa)$ where q is an operation token, u is the input interface, v the output interface, and $\kappa \in \mathbf{Kind}$ a kind drawn from a fixed finite set. Descriptors with the same (u, v, κ) remain distinct when their tokens differ. In **AgentHero**, q comprises the selected application/handler family and the complete **DagNode** value passed to it; it is not determined by kind. The set $\mathbf{Kind}$ is the fifteen-element enumeration **DagNodeKind**, and u, v are read from the node’s declared **inputs** and **outputs** together with the schema, if any, that the node’s tool declares.

Definition 3.7 (The skills operad). Let Σ_{AH} be the $\mathbf{C}_{\text{AH}}$ -coloured signature that has, for each operation descriptor $N = (q, u, v, \kappa)$ with canonical input enumeration $k_1 < \dots < k_n$ and each output name $k \in \text{dom}v$, one generator

$$\langle N, k \rangle \in \Sigma_{\text{AH}}(u(k_1), \dots, u(k_n); v(k)).$$

The skills operad is $\mathcal{O} = \text{Fr}(\Sigma_{\text{AH}})$, the free $\mathbf{C}_{\text{AH}}$ -coloured symmetric operad on Σ_{AH} . Its operations of arity n and output colour c are the composites of skills with n open input ports of the stated colours and one output port of colour c . Skills are operations of $\mathcal{O}$; port types are its colours. Substitution $\circ_i$ is Theorem 2.2: $f \circ_i g$ replaces the i -th open input port of f by the skill g that supplies it.

Remark 3.8. Theorem 3.7 splits a node with m declared outputs into m generators sharing a operation descriptor. This is forced: an operad’s operations have exactly one output. Coordinate functions can preserve deterministic correlations because they share the same input. The split still loses the fact that one runtime invocation produces all coordinates together.

3.3 Operads and multi-output structure

Theorem 3.8 shows that $\mathcal{O}$ cannot be the whole story, and it is better to say so now than to arrive at it as a surprise. The runtime’s nodes have several outputs; an operad’s operations have one. Two structures are therefore in play from here on, and the analysis is carried out for both.

Definition 3.9 (Coloured properad and PROP). A $\mathbf{C}$ -coloured properad $\mathcal{P}$ assigns to each pair of finite colour lists $(\vec{c}; \vec{d})$ a set $\mathcal{P}(\vec{c}; \vec{d})$ of operations with input colours $\vec{c}$ and output colours $\vec{d}$, carries commuting symmetric group actions on the inputs and on the outputs, and has a vertical

composition that grafts a nonempty set of outputs of one operation onto an equal number of inputs of another along a connected graph, subject to associativity, unit and equivariance axioms. A PROP is a properad with an additional unrestricted horizontal composition $\otimes$ that places two operations side by side, subject to the interchange law

$$(f_1 \otimes f_2) \circ (g_1 \otimes g_2) = (f_1 \circ g_1) \otimes (f_2 \circ g_2) \quad (2)$$

whenever both sides are defined. See Vallette [23] and Markl [16] for the full axiomatisations.

The hypothesis under test is stated for an operad, with colours the port types and operations the skills [2], so the operad has to be examined on its own terms. That is the reason not to work only with the PROP. The PROP is nevertheless needed to preserve a multi-output invocation as one syntactic operation. Table 1 records which results belong to which structure. The majority are shared, because they concern the validator and the scheduler rather than the algebra.

Concerns	Statements about $\mathcal{O}$	Statements about the PROP
Validator and scheduler	Theorem 5.1, Theorem 5.3, Theorem 5.10, Theorem 5.14, Theorem 5.6, Theorem 5.16	the same; these do not mention the algebraic structure
Rooted presentations	Theorem 3.12; general manifests require extra root and supplier data	unchanged, since grafting and layering are the same
Invocation structure	splits a multi-output node into coordinate generators	preserves a multi-output node as one operation
Order sensitivity	equivariance fails off the disjoint fragment, Theorem 5.16	interchange fails off the same fragment, Theorem 7.4

Table 1: Which results depend on which of the two candidate structures. Only the last two rows distinguish them.

Assumption 3.10 (Colour assignment). Throughout, a port that is not mentioned in any tool schema is assigned the colour $\top$ if it is a value port and **artifact** if it is an artifact port. This is the assumption that lets us speak of a colour alphabet at all for a runtime whose port names are open strings. It is not a fact about the code; it is the idealisation the operad reading requires, and Section 9 records what it hides.

3.4 Composites and manifests

Definition 3.11 (Rooted manifest presentation). A rooted manifest presentation consists of a finite acyclic dependency graph (V, E) , a distinguished output generator $r = \langle N, k \rangle$, and a partial supplier function from input slots in the backward slice of r to output generators of predecessors. Each supplied slot has exactly one supplier of a compatible colour. Its associated operation in $\mathcal{O}$ is obtained by recursively grafting those generators into their supplied slots; unsupplied slots remain the open inputs of the operation. A general manifest need not determine this data: it may have several terminal outputs, an outputless sink, or several possible suppliers for one name.

Lemma 3.12 (Well-definedness). *The operation associated to a rooted manifest presentation by Theorem 3.11 does not depend on the order in which the substitutions are performed.*

Proof. Substitutions at distinct input slots of a common node commute by Equation (1). Substitutions at nested slots compose associatively by the first partial-composition identity. Since E is acyclic, the substitution order can be taken to be any linear extension of E . Any two linear extensions differ by a

finite sequence of transpositions of E -incomparable pairs, and each such transposition is an instance of Equation (1). Hence the grafted tree is unique up to the isomorphism relation of Theorem 2.3. $\square$

Theorem 3.12 is a statement about the presentation data, not about every **AgentHero** manifest. The runtime does not require a distinguished output or a unique port-level supplier map. Whether execution respects a chosen presentation is separately constrained by Theorem 5.16.

3.5 Effect grading

Operations in a runtime are not pure. **AgentHero** records this explicitly, so we can model it rather than ignore it.

Definition 3.13 (Effect grade and independence). Let $\mathcal{E} = \{\text{pure}, \text{ro}, \text{wm}, \text{ee}, \text{ex}\}$ be the five effect classes pure, read-only, workspace-mutating, external-effect, and exclusive, and let $\mathcal{R}$ be a set of resource names. A node a has a class $e_a \in \mathcal{E}$ and a finite resource set $R_a \subseteq \mathcal{R}$. Distinct nodes $a \neq b$ are independent, written $a \# b$, when neither class is **ex** and either one class is **pure**, both are **ro**, or $R_a \cap R_b = \emptyset$.

The relation $\#$ is a Mazurkiewicz independence relation on node identifiers [17]: it is symmetric and irreflexive, and it is the relation under which two operations may be reordered. Theorem 3.13 is not an invention. It transcribes **DagConcurrencyClass** and **node_concurrency_compatible**, and Section 5 checks the transcription.

An independence relation carries a semantics of its own, and we use it rather than only stating L6 against it. Write Σ_M for the set of node identifiers of a manifest M , regarded as an alphabet. Let $\mathbb{M}(M)$ be the trace monoid $\Sigma_M^*/\approx$, where $\approx$ is the congruence generated by $ab \approx ba$ for every pair of nodes whose graded skills are independent [17]. An element of $\mathbb{M}(M)$ is an execution order up to the reordering of operations that cannot interfere.

Definition 3.14 (Effect word). The effect word of an execution of M is the sequence of node identifiers in the order in which their handlers are entered, read as an element of Σ_M^* , and its trace is the class of that word in $\mathbb{M}(M)$.

Theorem 5.8 below shows that the trace is determined by M alone even though, by Theorem 5.18, the final store is not. The effect layer quotients correctly where the data layer does not, which is the sharpest positive thing this paper has to say about the runtime's algebra.

Graded operations are the operations of an operad only if the grading is compatible with substitution, in the sense of a graded or parameterised effect system [8]. We do not claim **AgentHero** realises such a system, and we construct no Kleisli or graded algebra here; Section 9 says what would be needed.

3.6 The laws

Definition 3.15 (Realisation). A realisation of $\mathcal{O}$ is a tuple $(\mathcal{M}, I, \text{val}, \text{exec})$ where $\mathcal{M}$ is a set of manifests, $\text{val}: \mathcal{M} \rightarrow \{\text{ok}\} + \text{Err}$ is a validation function, $I(M)$ is the interface supplied by the initial store, and exec maps an accepted manifest and initial store to a final store.

Seven conditions separate semantic invariance from manifest-language obligations. L4 asks whether execution factors through the abstract operation. The other conditions concern validation, presentation, scheduling, arity, and effects; their failure does not violate an axiom of the free operad. Each condition is decidable by source inspection for the realisation examined here.

Family A, semantic invariance.

L4 *Equivariance*. If M and M' denote the same operation of $\mathcal{O}$ then $\text{exec}(M) = \text{exec}(M')$. In particular, transposing two siblings in the declaration order of M does not change the result.

Family B, well-posedness of the reading.

L1 *Input closure*. If $\text{val}(M) = \text{ok}$, every declared input slot is either present in $I(M)$ or has a unique compatible supplier among its ancestors.

L2 *Manifest identity representation*. For every colour c , the manifest language can express a node that copies a c -coloured input to a c -coloured output without changing it.

L3 *Schedule invariance*. The execution layering depends only on the dependency relation and declaration order, not on an auxiliary bracketing of a rooted presentation.

L5 *Arity determinacy*. The set of ports a node may read, and their colours, are determined by the node's signature and not by the composite in which the node occurs.

L6 *Effect independence*. If two nodes are scheduled concurrently by exec then their graded skills are independent in the sense of Theorem 3.13.

L7 *Well-foundedness*. If $\text{val}(M) = \text{ok}$ then its dependency relation is finite and acyclic; in particular no node depends on itself.

L1, L2, and L3 are manifest-language properties, not operad axioms. The free operad already contains identities and satisfies associativity. L1 asks whether validation establishes closure against a declared initial interface, L2 whether the manifest language represents the abstract identity, and L3 whether the scheduler ignores irrelevant presentation brackets. L5 is the condition under which Theorem 3.7 assigns a well-defined signature to a node considered on its own. L6 is the condition under which parallel composition is meaningful for effectful operations. L7 is the requirement for a finite execution order. Failures in Family B leave the algebra of the well-coloured fragment untouched; failures in Family A do not.

4 The system

4.1 Runtime

AgentHero's unit of work is a DAG manifest: a YAML or JSON document parsed into `DagManifest`, validated by `DagManifest::validate`, and executed by `DagExecutor<H>` against an application-supplied handler `H` implementing `NodeHandler`. The executor is deliberately domain-agnostic: it moves named JSON values and named artifact references between nodes and never interprets them.

Three crates carry the model. `dag-runtime` owns the manifest data types and validation. `dag-executor` owns the operation signature, the store, and the scheduler. `plugin-runtime` owns externally supplied capabilities and their lifecycle. A fourth, `app-sdk`, is the read path from an installed application's directory to an in-memory manifest.

4.2 The mapping

4.3 The operation signature

A skill is a function from an execution context to a result. The context carries the manifest, the node, a reference to the store, an attempt counter, a cancellation token, an optional durability receipt and an immutable execution world.

Model object	AgentHero identifier	Location at 1c3ad24
Port name (Theorem 3.1)	keys of <code>DagIo::values</code> , <code>DagIo::artifacts</code>	<code>dag-executor/src/lib.rs:76</code>
Colour <code>json[σ]</code> , artifact (Theorem 3.2)	<code>DagTool::input_schema</code> , <code>output_schema</code> ; artifact key namespace	<code>dag-runtime/src/lib.rs:492</code>
Operation descriptor (Theorem 3.6)	application handler plus the complete <code>DagNode</code> ; port signature from <code>inputs</code> , <code>outputs</code> , <code>kind</code>	<code>dag-executor/src/lib.rs:286</code> ; <code>dag-runtime/src/lib.rs:334</code>
Kind set <code>Kind</code>	<code>enumDagNodeKind</code> (15 variants)	<code>dag-runtime/src/lib.rs:109</code>
Skill (operation)	<code>traitNodeHandler::execute_node</code>	<code>dag-executor/src/lib.rs:286</code>
Operation input environment	<code>structNodeExecutionContext<'a></code>	<code>dag-executor/src/lib.rs:93</code>
Operation output	<code>structNodeExecutionResult</code>	<code>dag-executor/src/lib.rs:160</code>
Operation family per app	<code>traitDagApp (dag_type)</code>	<code>dag-executor/src/lib.rs:366</code>
Manifest dependency graph (partial data for Theorem 3.11)	<code>structDagManifest</code> , <code>structDagEdge</code>	<code>dag-runtime/src/lib.rs:658,636</code>
Validation <code>val</code>	<code>DagManifest::validate</code>	<code>dag-runtime/src/lib.rs:684</code>
Layering (L3)	<code>DagManifest::execution_layers</code>	<code>dag-runtime/src/lib.rs:1027</code>
Execution <code>exec</code>	<code>DagExecutor::execute_with_checkpoint</code>	<code>dag-executor/src/lib.rs:1601</code>
Store merge	<code>DagIo::merge</code>	<code>dag-executor/src/lib.rs:86</code>
Effect grade (Theorem 3.13)	<code>enumDagConcurrencyClass</code>	<code>dag-runtime/src/lib.rs:297</code>
Independence <code>#</code>	<code>node_concurrency_compatible</code>	<code>dag-executor/src/lib.rs:5485</code>
Composite identity	<code>manifest_hash</code>	<code>dag-executor/src/lib.rs:4652</code>
Capability linking	<code>resolve_bindings</code>	<code>plugin-runtime/src/lib.rs:1126</code>

Table 2: Model objects and the AgentHero identifiers that carry them at commit 1c3ad24. Supporting paths, identifiers and line ranges are collected in Section A.

```

/// App-side node dispatcher.
#[async_trait]
pub trait NodeHandler: Send + Sync {
    /// Execute one manifest node.
    async fn execute_node(
        &self,
        ctx: NodeExecutionContext<'_,>,
    ) -> anyhow::Result<NodeExecutionResult>;
}

```

Listing 1: The operation signature. `crates/dag-executor/src/lib.rs`, lines 284–292, AgentHero 1c3ad24.

```
#[derive(Debug, Clone, Default, PartialEq, Serialize, Deserialize)]
pub struct DagIo {
    /// Named JSON values.
    #[serde(default)]
    pub values: BTreeMap<String, serde_json::Value>,
    /// Named artifact references.
    #[serde(default)]
    pub artifacts: BTreeMap<String, ArtifactRef>,
}

impl DagIo {
    fn merge(&mut self, other: DagIo) {
        self.values.extend(other.values);
        self.artifacts.extend(other.artifacts);
    }
}
```

Listing 2: The port carrier and its merge, quoted without elision. `crates/dag-executor/src/lib.rs`, lines 75–90, AgentHero 1c3ad24.

The store that flows between operations is a pair of key-sorted maps.

Listing 2 carries the whole of Theorem 3.3 and half of Theorem 5.16. The key type is `String`, so the port-name set is open. The value type is `serde_json::Value`, so the carrier imposes no static constraint. And `merge` is `BTreeMap::extend`, whose documented behaviour on a duplicate key is to overwrite. The merge is therefore idempotent and associative but not commutative.

4.4 The kind vocabulary

```
pub enum DagNodeKind {
    PrepareInputs, Agent, Synthesizer, Verify, Gate,
    RenderArtifacts, ModerationReady, IngestSource, Tool,
    Artifact, DagCall, Loop, Branch, Map, Approval,
}
```

Listing 3: The closed kind vocabulary. `crates/dag-runtime/src/lib.rs`, lines 107–125, AgentHero 1c3ad24.

The fifteen kinds are a closed set, unlike the port names. Five of them, `Gate`, `Loop`, `Branch`, `Map` and `Approval`, are handled by the executor itself rather than dispatched to the handler; the predicate that separates them is `is_ordinary_handler_node` at `dag-executor/src/lib.rs:6340`. Those five are the derived operations of Section 6.

4.5 A worked manifest

Listing 4 exposes the gap between a dependency edge and operadic substitution. The first node has output `result.json` and the second consumes that name, but the second declares no output. It therefore supplies no distinguished generator for the root required by Theorem 3.11. The edge imposes execution order; it does not by itself determine a single operation of the one-output operad.

The name `result.json` occurs in both node declarations, and it matches. Nothing in `DagManifest::validate` requires that it match. Section 5 establishes this.

```

nodes:
- id: write_policy_report
  kind: tool
  tool: write_policy_report
  outputs: [result.json]
  required: true
- id: human_checkpoint
  kind: tool
  tool: human_checkpoint
  inputs: [result.json]
  required: true
edges:
- from: write_policy_report
  to: human_checkpoint

```

Listing 4: A complete two-node manifest. `agenthero/apps/platform-smoke/dags/tool-policy-smoke.yaml`, lines 20–33, AgentHero 1c3ad24.

4.6 Schema discipline

Two functions in the executor apply a declared schema at dispatch time. The first restricts the store to a node’s declared inputs before checking; the second checks a node’s produced artifacts against its declared outputs.

```

fn contract_input_io(node: &DagNode, inputs: &DagIo) -> DagIo {
  if node.inputs.is_empty() {
    return inputs.clone();
  }

  let mut allowed: BTreeSet<&str> = node.inputs.iter().map(String::as_str).collect();

```

Listing 5: The early return that makes a node’s readable port set depend on its context. `crates/dag-executor/src/lib.rs`, lines 4367–4372, AgentHero 1c3ad24. The function continues to line 4405, where it filters the store to the allowed names; that part is described in the text and not quoted.

Listing 5 is the source of Theorem 5.14: the early return on the empty case means a node that declares no inputs receives the entire accumulated store. Listing 6 is the one genuine colour check on the output side, and it is a containment, not an equality: a node may return fewer artifacts than it declares.

5 Checking the laws

Only L4 depends on whether nodes are read in the operad or the PROP; the remaining conditions concern validation and scheduling.

5.1 Law L7: finite dependency order

Code observation 5.1 (Shape validation). Let M be a manifest with $\text{validate}(M) = \text{Ok}$ at commit 1c3ad24. Then

1. node identifiers, role identifiers and tool identifiers are pairwise distinct;

```

fn validate_node_artifact_output_contract(node: &DagNode, outputs: &DagIo) -> anyhow::Result<()>
{
    let declared: BTreeSet<&str> = node.outputs.iter().map(String::as_str).collect();
    for key in outputs.artifacts.keys() {
        if !is_safe_artifact_key(key) {
            anyhow::bail!("unsafe artifact output key '{key}' returned by node '{node.id}'");
        }
        if !declared.contains(key.as_str()) {
            anyhow::bail!("undeclared artifact output '{key}' returned by node '{node.id}'");
        }
    }
    Ok(())
}

```

Listing 6: The artifact output contract. `crates/dag-executor/src/lib.rs`, lines 4336–4353, AgentHero 1c3ad24.

2. every endpoint of every edge, every gate source, every branch target and every `parent_node` reference resolves to a declared node;
3. every node reference to a role or tool resolves to a declared role or tool, and every role kind is a member of the manifest’s `accepts` list;
4. the `parent_node` relation is acyclic;
5. the dependency relation induced by `edges` is acyclic;
6. for each node of kind `Gate`, `Loop`, `Branch`, `Map` or `Approval`, the matching policy is present and its numeric bounds are nonzero.

Source inspection. Items 1 to 3 and 6 are the explicit checks in `DagManifest::validate` (`dag-runtime/src/lib.rs:685–906`), which returns `DuplicateNode`, `DuplicateRole`, `DuplicateTool`, `MissingNode`, `MissingRole`, `MissingTool`, `KindNotAccepted` and the five `...NodeMissingPolicy` and `...NodeInvalidPolicy` variants respectively. Item 4 is the explicit ancestry walk at lines 1004 to 1022, returning `ParentNodeCycle`. Item 5 holds because `validate` ends with `self.execution_layers().map(|_|())` at line 1024. `execution_layers` (lines 1027 to 1083) performs a layered topological peel that returns `DagError::Cycle` whenever the set of nodes with no remaining dependencies is empty and the remaining set is not. $\square$

Corollary 5.2 (L7). *Assume Theorem 5.1. Then L7 holds for the realisation of Table 2: every validated manifest has a finite acyclic dependency relation.*

Proof. Finiteness follows from the finite node and edge lists. Acyclicity is Theorem 5.1(5). This establishes an execution order, not a unique operation of $\mathcal{O}$; the latter requires the additional presentation data of Theorem 3.11. $\square$

5.2 Law L3: schedule invariance

Code observation 5.3 (Layering depends only on the dependency relation). Let M be a validated manifest, D its direct-dependency relation and $\prec$ the declaration order on nodes. Then `execution_layers` returns the sequence $L_0, L_1, \dots$ defined by

$$L_0 = \{N : D^{-1}(N) = \emptyset\}, \quad L_{j+1} = \left\{ N \notin \bigcup_{i \leq j} L_i : D^{-1}(N) \subseteq \bigcup_{i \leq j} L_i \right\},$$

with each L_j listed in $\prec$ order. In particular the sequence is a function of $(D, \prec)$ alone.

Source inspection. The implementation builds `remaining`, a map from each node to the set of its direct predecessors, from `edges` alone (lines 1035 to 1059). It then repeatedly collects the nodes with empty remaining sets, sorts them by the index of the node in `manifest.nodes` (`node_order`, line 1029), removes them, and subtracts them from the remaining sets of all other nodes (lines 1061 to 1080). This is exactly the recursion displayed, and it reads no other field of the manifest. Adaptive worker templates are excluded before the loop begins, which is a restriction of the node set, not of the recursion. $\square$

Proposition 5.4 (L3). *Assume Theorem 5.3. Then L3 holds: two manifests with the same dependency relation and declaration order induce the same layering. Auxiliary brackets in a rooted presentation do not change that layering.*

Proof. The layering is a function of $(D, \prec)$ alone. Bracketing is not an input to the function. $\square$

Remark 5.5. Theorem 5.4 is a scheduler result, not a proof of operad associativity or semantic equality. The result may still depend on order within a layer, as Theorem 5.16 shows.

5.3 Law L6: checked independence

Code observation 5.6 (L6). At commit 1c3ad24, if `DagExecutor` schedules a layer concurrently then the graded skills of its nodes are pairwise independent in the sense of Theorem 3.13. That is, L6 holds for this realisation.

Source inspection. `can_execute_layer_concurrently` (`dag-executor/src/lib.rs:5333-5358`) returns true only when its final clause holds: `node_concurrency_compatible` holds for every unordered pair of nodes in the layer. That function (lines 5485 to 5502) returns false if either grade is `Exclusive`, returns true if either grade is `Pure` or both are `ReadOnly`, and otherwise returns the negation of `concurrency_resources_overlap`. Comparing this with Theorem 3.13 clause by clause gives the stated equivalence. The grade of a node is its declared `DagNodeConcurrency::class` when present. Otherwise it is the value computed by `derived_concurrency_class` (lines 5406 to 5444) from the node kind and, for tool nodes, from the tool’s filesystem write policy. $\square$

Remark 5.7 (The qualification). When the pairwise-compatibility test fails, the executor does not reject the manifest. It falls through to a sequential path (`dag-executor/src/lib.rs:1804-2190`) that executes the layer’s nodes one at a time in declaration order, merging each node’s outputs into the store before taking the next node’s snapshot. So interference is resolved by choosing an order rather than by refusing the composite. That is a defensible engineering decision and an unsound operadic one. The composite’s meaning becomes a function of $\prec$ whenever a layer is not independent.

Proposition 5.8 (The trace of an execution is well defined). *Assume Theorem 5.3 and Theorem 5.6. Let M be a validated manifest all of whose layers are scheduled concurrently. Then every execution of M has the same trace in $\mathbb{M}(M)$, and that trace is determined by the dependency relation of M alone. In particular the trace is invariant under transposition of layer siblings in the declaration order, whether or not their output keys are disjoint.*

Proof. By Theorem 5.3 the layer sequence $L_0, L_1, \dots$ is a function of the dependency relation and the declaration order, and every execution enters the handlers of L_j before those of L_{j+1} . Two executions can therefore differ only in the order of entry within a single layer. Their effect words differ by a permutation of the letters of each layer, that is, by a finite sequence of transpositions of

letters lying in a common layer. By Theorem 5.6 any two distinct nodes of a concurrently scheduled layer have independent graded skills. Each such transposition is therefore one of the generating relations of $\approx$, and the two words are equal in $\mathbb{M}(M)$. Since the layer sequence depends on the declaration order only through the order within layers, and that order is quotiented away, the trace depends on the dependency relation alone. $\square$

Remark 5.9. Theorem 5.8 and Theorem 5.18 are the two halves of the same picture and they point in opposite directions. The order in which effects happen is well defined as an element of the trace monoid, for every validated manifest with concurrently scheduled layers. The value the run produces is not well defined as an element of the store set, unless the sibling output keys are disjoint. A scheduler can therefore be sound about what it does and unsound about what it computes, and AgentHero at this commit is exactly that.

5.4 Law L1: input closure

Code observation 5.10 (Partial colour discipline). At commit 1c3ad24, the following hold. Items 2 and 3 are the fragments of a colour discipline that are present; items 1 and 4 are what is absent.

1. *No static check.* The field `DagNode::inputs` is read by no branch of `DagManifest::validate`. Consequently a manifest in which some node declares an input name that no node anywhere declares as an output validates successfully.
2. *Artifact output containment.* For every node of a validated manifest, every artifact key the node returns lies in the node’s declared `outputs` and is a safe relative path, or the attempt fails.
3. *Tool-scoped schema checking.* If a node names a tool and that tool declares `input_schema`, the store restricted to the node’s declared input names is checked against that schema before dispatch; symmetrically for `output_schema` after a successful dispatch. If the node names no tool, or the tool declares no schema, no check occurs.
4. *Values are unchecked.* No analogue of item 2 exists for `DagIo::values`: a node may return value keys it did not declare.

Source inspection. Item 1: the token `inputs` occurs four times in `crates/dag-runtime/src/lib.rs`, at line 130 (a display string for the `PrepareInputs` kind), line 346 (the field declaration on `DagNode`), line 1087 (a documentation comment) and line 1189 (a field of an unrelated structure). None is inside `validate`, whose body spans lines 684 to 1025. The validation clauses that do concern port names concern outputs only: `for output in &node.outputs { if !is_safe_artifact_key(output) ... }` at lines 810 to 818.

Item 2: `validate_node_artifact_output_contract` (`dag-executor/src/lib.rs:4336-4353`), quoted as Listing 6, is called from the node dispatch path at line 3797. The guard at lines 3793 to 3795 requires the normalised status to be `Ok` or `Degraded`, and `validate_node_output_contract` is called at line 3798. The function bails on any artifact key that is unsafe or undeclared.

Item 3: `validate_node_input_contract` (lines 4309 to 4321) obtains the pair `(tool, schema)` through `node_tool_schema` (lines 4355 to 4365), which returns `None` when the node has no tool or the tool has no schema. It then calls `validate_dag_io_schema` on `contract_input_io(node, inputs)`. The output side is `validate_node_output_contract` (lines 4323 to 4334), structured identically.

Item 4: the only call sites of `validate_dag_io_schema` are the two in item 3, and the only containment check on returned keys is item 2, whose loop ranges over `outputs.artifacts.keys()` and not over `outputs.values.keys()`. $\square$

Example 5.11 (The colour set is open). Take Listing 4, choose an initial-store interface containing `result.json` but not `result1.json`, and rename the consumer's declared input to `result1.json`. Every clause of Theorem 5.1 still holds, so `validate` returns `Ok`. The accepted manifest has an input supplied by neither the initial interface nor an ancestor. The free operad can treat that slot as an open input; the failure is instead that validation does not establish the closure condition L1 requires.

Corollary 5.12 (Validation does not establish input closure). *Assume Theorem 5.10(1). Then `val` accepts manifests with an input absent from both $I(M)$ and every ancestor output, so L1 fails for this realisation.*

Proof. The renamed input in Theorem 5.11 is absent from the chosen initial interface and from all ancestor outputs, yet the source inspection in Theorem 5.10(1) shows that validation does not examine it. $\square$

Remark 5.13. The result does not challenge the axioms of $\mathcal{O}$. It says the manifest validator does not prove that a run is closed relative to its declared initial interface.

5.5 Law L5: composite-dependent reads

Code observation 5.14 (Arity is not determined by the node). At commit 1c3ad24, L5 fails. There is an operation descriptor N and two validated manifests M, M' both containing N such that the arity of the operation N contributes to the composite of M differs from the arity it contributes to the composite of M' .

Source inspection. Let N be any node with `inputs` absent, that is with `node.inputs` empty; the field carries `#[serde(default)]` (`dag-runtime/src/lib.rs:345-346`), so this is the default and it is common in practice. The environment such a node receives at dispatch is `contract_input_io(node, inputs)` for the schema check and the full `inputs` for the handler. `contract_input_io` returns `inputs.clone()` unrestricted when `node.inputs.is_empty()` (Listing 5, lines 4368 to 4370). The store passed as `inputs` is the accumulated store, which by the layer loop (`dag-executor/src/lib.rs:1627-1636, 1751`) contains the initial input together with every value and artifact merged by every node completed in an earlier layer. Take M to be the manifest containing N alone, and M' to be M extended by one unrelated predecessor node that emits one value key. The store N observes has one more key in M' than in M , so the arity of the operation contributed by N differs between the two. $\square$

Remark 5.15. Theorem 5.14 is not a defect report. A store-passing scheduler that hands each node the whole accumulated environment is a reasonable design, and it is what most workflow engines do. The point is narrower: the design is not operadic, because in an operad an operation's arity is part of the operation. The fragment on which L5 does hold is exactly the set of manifests in which every node declares its inputs explicitly, and Listing 5 shows that the runtime already distinguishes that fragment. Requiring nonempty `inputs` would recover L5 at the cost of verbosity.

A second reading is available and is better than the one L5 tests. A node that reads the ambient store is naturally modelled as an operation with an implicit environment parameter, in the manner of a reader effect. The operadic arity is then the declared input list, and the environment is carried by the grading of Theorem 3.13. On that reading L5 is not violated, because the environment was

never claimed to be part of the arity. The consequence survives the change of reading, though. The environment a node receives is a function of the entire composite. Two manifests that are equal as elements of $\mathcal{O}$ can therefore supply a node with different environments, and the executor's output map still does not factor through $\mathcal{O}$. Theorem 5.18 reaches the same conclusion from an independent direction, so the conclusion does not depend on which of the two readings one prefers.

5.6 Law L4: equivariance

Code observation 5.16 (Order sensitivity of parallel composition). Let M be a validated manifest and let N_1, N_2 be two nodes occurring in the same execution layer L_j , with $N_1 \prec N_2$. Let M' be M with the two transposed in `manifest.nodes` and otherwise unchanged, so that M and M' denote the same operation of $\mathcal{O}$. Suppose the layer is scheduled concurrently. Then

1. if the actual stores returned by N_1 and N_2 have disjoint key domains, the store after layer L_j is the same for M and M' ;
2. if the two nodes return a common key k with distinct values, the store after layer L_j differs: it carries N_2 's value for k under M and N_1 's value under M' .

Thus actual output disjointness is sufficient for invariance, while a collision with distinct values is a counterexample. Declared output disjointness alone is insufficient because value handlers may return undeclared keys.

Source inspection. Both nodes receive the same argument: the executor takes `letsnapshot=output s.clone()` once, before dispatching the layer (`dag-executor/src/lib.rs:1718`), and passes `&snapshot` to `execute_ordinary_nodes_concurrently` (line 1724), which clones it per node (line 2372). So neither node's produced value depends on the other, and transposition does not change the two produced stores.

The order in which the produced stores are merged does change. `execution_layers` sorts each ready set by `node_order`, the index of the node in `manifest.nodes` (`dag-runtime/src/lib.rs:1070`). `execute_ordinary_nodes_concurrently` tags each spawned task with its index in that list and reassembles the outcomes into an index-ordered vector (`dag-executor/src/lib.rs:2366-2400`). The layer loop then merges in that order, `outputs.merge(produced)` (line 1751).

`DagIo::merge` is `BTreeMap::extend` on both components (Listing 2). On a key present in only one map, extend inserts it; on a key present in both, extend overwrites with the value from the argument. Therefore for disjoint domains the two merge orders give the same map, which is item 1. For a shared key k the surviving value is the one from the map merged last, which is N_2 's under M and N_1 's under M' , which is item 2. $\square$

Example 5.17 (Merge collision). Two verification nodes in one layer, each returning a value under the key `status`, one "ok" and one "degraded". Downstream nodes read `status`. Reordering the two declarations in the manifest changes what they read. Nothing in `validate` detects the collision, because `outputs` is checked for path safety and not for disjointness across siblings.

Corollary 5.18. *Assume Theorem 5.16(2). Then the map sending a validated manifest to its final store does not factor through $\mathcal{O}$, and there is no algebra of $\mathcal{O}$ in the sense of Theorem 2.4 whose action is the executor's behaviour on all validated manifests.*

Proof. By the premise there are two manifests denoting the same operation of $\mathcal{O}$ with distinct final stores. A factorisation through $\mathcal{O}$ would assign them the same value. $\square$

Remark 5.19. Declared disjointness is checkable from a manifest, but it would guarantee L4 only if validation also confined returned value keys to each node’s declaration. Artifact outputs already satisfy that containment; value outputs do not.

5.7 Law L2: manifest identity representation

Code observation 5.20 (No manifest-level identity). At commit `1c3ad24`, L2 fails. No member of `DagNodeKind` denotes a generic identity node, and the executor supplies no identity behaviour for any kind. The abstract identities of the free operad still exist.

Source inspection. The fifteen kinds are listed in Listing 3. Five are executor-handled combinators (`Gate`, `Loop`, `Branch`, `Map`, `Approval`), each with a policy that `validate` requires to be present and in range (Theorem 5.1(6)), so none is an identity. The remaining ten are ordinary handler nodes by `is_ordinary_handler_node` (`dag-executor/src/lib.rs:6340-6349`) and are dispatched to `N odeHandler::execute_node`. The executor does not constrain that behaviour beyond the output contracts of Theorem 5.10(2,3). In particular `PrepareInputs`, whose name suggests a unit, is an ordinary handler node with no executor-supplied semantics. $\square$

Remark 5.21. An application may of course implement an identity handler, and `GenericToolRunner` (`dag-executor/src/lib.rs:387`) is close to one in spirit: its documentation says it “deliberately knows how to invoke tools and record artifacts, but not how to interpret domain-specific results”. But a unit that exists only inside an application handler is not a generic identity represented by the manifest language.

5.8 Summary

Law	Content	Verdict	Where established
L1	input closure	fails	Theorem 5.10 and Theorem 5.12: initial or ancestor supply is not checked
L2	manifest identity representation	fails	Theorem 5.20: no generic identity kind
L3	schedule invariance	holds	Theorem 5.4 via Theorem 5.3
L4	equivariance	conditional	Theorem 5.16: actual disjoint outputs suffice; a distinct-value collision is a counterexample
L5	ports a node may read fixed by its signature	fails	Theorem 5.14: empty <code>inputs</code> admits the whole store
L6	independence of concurrent nodes	holds	Theorem 5.6; qualified by Theorem 5.7
L7	finite acyclic dependency order	holds	Theorem 5.2 via Theorem 5.1

Table 3: Verdicts for the seven laws against `AgentHero` at commit `1c3ad24`. Each verdict is a statement about the source at that commit, established by inspection.

Three of seven hold, one has a sufficient fragment and a counterexample, and three fail. L4 is the semantic invariance condition; the others are manifest-language and scheduler obligations. That is the content of the phrase “skills are operad operations” for this system. One word in Table 3 should not be read charitably. “Conditional” for L4 is not a partial pass. By Theorem 5.18 the executor is not an algebra of $\mathcal{O}$ on all validated manifests. The conditional verdict records a sufficient fragment and a concrete counterexample; it does not assert closure of that fragment under substitution.

Some of these findings needed the formalism and some did not. L1 did not. That **AgentHero** is dynamically typed and performs no static port check is visible directly from Listing 2; Theorem 5.12 states it precisely but adds little. The other findings are not of that kind. Theorem 5.16 identifies a condition, layer-wise disjointness of sibling output keys, that is checkable in linear time from a manifest and that no reading of the code as “dynamically typed” would suggest looking for. Theorem 7.4 shows the same condition governs the richer structure, so it is not an artifact of the choice of $\mathcal{O}$. Theorem 7.2 separates deterministic correlation from nondeterminism. Theorem 5.8 finds a level at which the runtime is sound, and does so by exhibiting the monoid rather than by asserting soundness. Theorem 6.7 locates the missing check already implemented one layer up in the same repository, which turns a diagnosis into a concrete remedy. The formalism earns its place on those five, not on L1.

6 Derived operations

Banu [2] names three composition operations for the Skills pillar: serial, parallel and trace. **AgentHero** has five executor-handled combinators. We ask, for each, whether it is a new primitive of the operad or a derived operation, that is, definable from substitution plus data already present.

6.1 Gate and approval as guards

Code observation 6.1 (Guards are partial operations). **DagGate** and **DagApproval** are derived: each denotes the restriction of an operation to a sub-domain of its input store, and neither introduces a composition law.

Source inspection. **DagGate** (`dag-runtime/src/lib.rs:180-186`) carries `min_usable:Option<u32>` and `sources:Vec<String>`. `validate` requires each source to name a declared node and `min_usable` to be nonzero when present (lines 819 to 828 and 988 to 994). **DagApproval** (lines 254 to 262) carries `approved_key` and an optional `grant_key`, and `validate` requires the keys to be nonempty (lines 872 to 884). In both cases the node’s effect is to admit or withhold the continuation depending on a predicate evaluated against the store. The derived concurrency class of both is **Pure** (`dag-executor/src/lib.rs:5427-5431`). A predicate on the store restricting an operation’s domain is the partial-operation construction, available in any operad whose operations are partial functions, and it adds no new γ . $\square$

6.2 Branch as an indexed family

Code observation 6.2 (Branch selects a composite). **DagBranch** is derived: it denotes a finite family of composites indexed by the value of a key, together with a selector, and not an operation of $\mathcal{O}$.

Source inspection. **DagBranch** (`dag-runtime/src/lib.rs:199-206`) carries `decision_key:String`, `cases:BTreeMap<String,Vec<String>>` and `default:Vec<String>`. `validate` requires the decision key to be nonempty and at least one of `cases` and `default` to be nonempty. Every case label must be nonempty with a nonempty target list (lines 837 to 854), and every target must name a declared node (lines 995 to 1001). At execution the branch node computes a selection from the store snapshot and records it in `branch_selections` (`dag-executor/src/lib.rs:1868-1911`). Unselected descendants are skipped by `is_unselected_by_branch`. So the manifest declares $|\text{cases}| + 1$ sub-composites and the runtime picks one. Formally this is a map from a decision value

to a choice of element of $\mathcal{O}$, which lives in the set of families indexed by the decision alphabet, not in $\mathcal{O}$ itself. $\square$

6.3 Map as bounded fan-out

Code observation 6.3 (Map is a bounded parallel family). `DagMap` is derived: it denotes, for each $n \leq \text{max_items}$, the n -fold parallel composite of a single sub-operation, with n determined at dispatch.

Source inspection. `DagMap` (`dag-runtime/src/lib.rs:232-244`) carries `items_key`, `item_key`, `index_key`, an optional `item_id_key`, a mandatory `max_items` and an optional `max_concurrency`. `validate` requires all the key strings to be nonempty and both bounds to be nonzero (lines 855 to 871). At execution `execute_map_node` (`dag-executor/src/lib.rs:2654`) reads the item list from the store under `items_key` and fails when its length exceeds `max_items` (lines 2700 to 2706). Each iteration is dispatched with the item bound under `item_key` and the index under `index_key`, both of which `contract_input_io` adds to the allowed input set for map nodes (lines 4379 to 4387). So the arity of the parallel composite is read from the store at dispatch and bounded statically. This is a bounded family of operations, not one operation, for the same reason as Theorem 6.2, and it is derived because each member is an ordinary n -fold substitution. $\square$

6.4 Bounded loops

Code observation 6.4 (Loop is not a categorical trace). `DagLoop` is derived and is a bounded unrolling. It is not a trace in the sense of a traced monoidal category [7]: no fixed point is computed, no fixed point is required to exist, and the number of iterations is a manifest constant.

Source inspection. `DagLoop` (`dag-runtime/src/lib.rs:188-193`) carries `max_rounds:u32` and a `continue_key` defaulting to `"loop_continue"`. `validate` requires `max_rounds` to be nonzero (lines 829 to 836) and rejects a `retry` policy on a `Loop` node (lines 885 to 895). `execute_loop_node` (`dag-executor/src/lib.rs:2415-2652`) initialises `visible_inputs` from the snapshot (line 2439). On each round it dispatches the node with `round_inputs` cloned from `visible_inputs` (line 2446), then merges the produced store into both `visible_inputs` and `combined_outputs` (lines 2499 to 2503). It stops early when the value at `continue_key` is absent or not the boolean `true` (lines 2476 to 2485). The loop therefore computes the r -th iterate for some $r \leq \text{max_rounds}$ and returns it. A trace requires a canonical solution of a feedback equation, and there is none here: the iterate is returned whether or not it is a fixed point, and no convergence check is performed. Unrolling a bounded feedback edge is a finite composite of substitutions, hence derived. $\square$

Remark 6.5. Traces do exist for recursive computation once the ambient category carries enough structure: in a cpo-enriched traced monoidal category the feedback operator is a least fixed point and the finite unrollings are its approximants [6]. The obstruction at commit `1c3ad24` is not the boundedness but the absence of that structure. The runtime places no order on `DagIo`, imposes no monotonicity condition on `NodeHandler` implementations, and returns the r -th iterate as the answer rather than as an approximant of a limit. Supplying the missing hypothesis is listed as an open problem in Section 12.

Remark 6.6. This is the point at which Banu's third named operation, $\text{Tr}(A)$, does not transfer. What the code offers is $A^{(r)}$ for r bounded by a manifest constant, with an early-exit predicate. Calling that a trace overstates it. Bounded unrolling with an exit predicate is the ordinary operational choice for feedback in a scheduler, since it is what rules out divergence. Nothing here counts it

against the runtime. What is withheld is only the categorical name, for the reason Theorem 6.5 gives. A statement that would be defensible on this evidence is that the runtime supports bounded iteration with an application-supplied continuation predicate; Listing 7 is the declaration form.

```
- id: lean_review_fix_code
  kind: loop
  tool: review_fix_code
  loop:
    max_rounds: 3
  inputs:
    - review_loop/proof_obligations.json
    - review_loop/lean_targets.json
  outputs:
    - review_loop/lean/env/env_result.json
```

Listing 7: A loop node with a manifest-fixed bound. `agenthero/apps/grokrxiv/dags/review-loop.yaml`, lines 158–167 of a node spanning lines 158–175; seven further output keys follow. AgentHero 1c3ad24.

6.5 Sub-DAG invocation

`DagNodeKind::DagCall` is the kind whose name suggests operadic substitution of a whole sub-manifest. At commit 1c3ad24 the executor does not expand it. `DagCall` appears in `dag-executor/src/lib.rs` only at line 5439, where it is assigned a derived concurrency class, and at line 6732, where its `dag_type` is used as a report label. It is an ordinary handler node by `is_ordinary_handler_node`, so expanding a sub-manifest is the application’s responsibility, not the runtime’s. The nesting that would make substitution literal in the code is therefore delegated, and the executor sees a leaf where the model sees an internal vertex.

6.6 Enforced colour discipline

Code observation 6.7 (Capability linking). The linking check that L1 requires and `DagManifest::validate` omits is present one layer above the DAG, on plugin capabilities.

Source inspection. `PluginInstanceSpec` (`plugin-runtime/src/lib.rs:500–509`) declares `requires:Vec<CapabilityRequirement>` and `provides:Vec<String>`, where a `CapabilityRequirement` (lines 480 to 485) is a capability key together with an optional flag. `PluginInstanceSpec::validate` (lines 511 to 549) requires every key on both sides to satisfy `valid_capability_key` (lines 557 to 563) and requires both lists to be duplicate-free. `resolve_bindings` (lines 1126 to 1164) then collects, for each requirement, the active instances whose `provides` list contains the required key. It sorts the candidates for determinism, binds the first, and returns `None` when no provider exists and the requirement is not optional. At the activation site (lines 939 to 956) a `None` makes the candidate be skipped by a `continue`, so an instance with an unsatisfied non-optional requirement never has its state set to `Activating`. That is exactly the colour-matching check of L1, performed by name equality on capability keys. $\square$

Remark 6.8. Theorem 6.7 sharpens the paper’s overall finding. The repository does contain a working port-matching discipline; it is applied to which capabilities a plugin may assume, not to which values a node may read. The two are the same construction at different granularities, which suggests the gap in L1 is an omission rather than an architectural obstacle.

7 Multi-output nodes

Multi-output nodes require a PROP for a faithful operation signature. That change does not repair the runtime’s order sensitivity.

Remark 7.1 (The definitional half). No operation of a coloured operad has more than one output; that is part of Theorem 2.1. So the splitting in Theorem 3.7 is forced rather than chosen, and nothing is learned by observing it. The substantive question is what the splitting costs; the answer is not definitional.

Theorem 7.2 (Set-valued algebras are deterministic). *Let N be an operation descriptor with outputs $k_1, \dots, k_m$, and let A be an algebra of $\mathcal{O}$ in **Set**. If A_j is the function assigned to $\langle N, k_j \rangle$, then a fixed input tuple x determines the single output tuple $(A_1(x), \dots, A_m(x))$. If the intended semantics permits two distinct output tuples at the same fixed input, no such algebra represents it. This obstruction is nondeterminism, not correlation among coordinates.*

Proof. An algebra in **Set** assigns a function to each generator. Evaluating those functions at the same x produces one tuple. Shared inputs can still correlate coordinates; for example $A_1(x) = A_2(x) = x$ has diagonal image. $\square$

Example 7.3 (A nondeterministic node). A node of derived class **ExternalEffect** may perform a model call or process invocation (Theorem 5.6). Repeated executions with the same recorded input can return distinct pairs of outputs. A **Set**-valued algebra cannot represent that behaviour without adding the relevant world state to the input or changing the semantic category.

7.1 Multi-output structure

Passing from $\mathcal{O}$ to the PROP of Theorem 3.9 represents a multi-output node as one syntactic operation. A node with output interface v becomes a single operation with output colour list $(v(k_1), \dots, v(k_m))$. A **Set**-valued algebra remains deterministic, as Theorem 7.2 shows. The change also does not remove the failure recorded in Theorem 5.16, which reappears in a new place.

Proposition 7.4 (Interchange fails under the same hypothesis). *Assume Theorem 5.16. Read the nodes of a validated manifest as operations of a $\mathbf{C}_{\text{AH}}$ -coloured PROP, with horizontal composition given by placing two nodes in one execution layer and vertical composition given by the layer order. Interpret a horizontal composite by dispatching both operations from one snapshot and merging their returned stores in declaration order. Let g_1, g_2 be layer siblings and f_1, f_2 siblings in the next layer. Suppose each f_i declares and reads only keys returned by g_i , and the actual returned key domains of g_1 and g_2 are disjoint. Then Equation (2) holds for these four operations. If the two domains collide, there are handlers for which it fails.*

Proof. By the first paragraph of the proof of Theorem 5.16, all members of one layer are dispatched against a single frozen store. The two produced stores of g_1 and g_2 therefore do not depend on each other or on the bracketing. The two sides of Equation (2) differ only in what store is visible to f_1 and f_2 . On the left the executor merges both produced stores into the accumulated store before the next layer; on the right each strand carries only its own.

Suppose the returned domains are disjoint. By Theorem 5.16(1) the merged store agrees with each strand’s store on that strand’s keys, and by Listing 5 an f_i that declares and reads only its strand’s keys sees the same values. Hence each f_i receives the same argument on both sides and the two composites agree.

Suppose instead that g_1 and g_2 both write a key k with distinct values, and let f_1 be a handler that returns the value it finds at k . On the left f_1 receives the value written by whichever of g_1, g_2 appears later in `manifest.nodes`, by Theorem 5.16(2); on the right it receives g_1 's. The two composites differ. $\square$

Remark 7.5. The PROP preserves a multi-output node as one syntactic operation. It does not make an effectful node deterministic, and it does not remove the shared-store collision that breaks interchange.

Remark 7.6 (What the interchange failure diagnoses). It would be a misreading to present Theorem 7.4 as a surprising structural discovery. A monoidal product asks that two operations act on disjoint tensor factors, and **AgentHero**'s layer siblings act on one shared store which they may both write. The predictable consequence of putting a shared-memory program into a strict tensor is that interchange fails, and that is what happens. The value of the calculation is not the failure but its exact boundary. Interchange holds under actual output disjointness and strand-local reads. Declared disjointness would be a checkable sufficient condition only if returned value keys were also confined to declarations. These conditions mark the fragment on which the shared store behaves like a tensor product of disjoint factors, and it is the same fragment Theorem 5.16 isolates for the operad. A reader who prefers to start from a symmetric monoidal category of state transformers, or from a graded effect monad in the sense of [8], will find the same fragment there. The fragment is fixed by the merge, not by the ambient structure.

Example 7.7 (Multi-output in practice). The node `lean_review_fix_code` of Listing 7 declares eight output keys (`agenthero/apps/grokrxiv/dags/review-loop.yaml`, lines 166 to 174), of which the excerpt shows one. Under Theorem 3.7 that node contributes eight independent generators, and their production by one invocation is not recorded anywhere in $\mathcal{O}$.

8 Layer composition

8.1 Layer contribution

The Memory layer supplies state that persists across invocations. The Skills layer adds *composability of work*: a named unit with a declared interface that can be substituted into another unit's input slot. It also adds a validation function that accepts or rejects the substitution before anything runs. Table 3 says which parts of that promise the evidence supports: the shape of a composite is checked (L7, L3), its independence structure is checked (L6), and its typing is checked only where a tool declares a schema (L1).

8.2 Lower-layer composition

A skill may read and write memory. In the model this is the statement that the colour **Mem** of Part I [10] may occur among the $u(k_i)$ and $v(k)$ of a node signature. Two consequences follow. An operation whose input colour is **Mem** is not pure in the sense of Theorem 3.13, so L6 constrains where it may be scheduled. `derived_concurrency_class` assigns **ReadOnly** to **Verify** nodes and **WorkspaceMutation** to **RenderArtifacts**, **ModerationReady**, **IngestSource** and **Artifact** nodes (`dag-executor/src/lib.rs:5427-5436`), which is the mechanism by which memory-touching operations are kept out of one another's way. The composition is also architectural rather than shipped. At commit 1c3ad24 **AgentHero** has no dependency on **ContextFS**, so a node that reads Part I's carrier would do so through a tool, and the model does not depend on any such tool existing.

8.3 Upper-layer composition

A wiring in Part III’s sense [11] can carry a skill at a box. The relation between $\mathcal{O}$ and $\mathcal{W}$ is settled in [11] and not here. They are distinct operads with different concrete colour sets, related by a type erasure whose section is not checkable at these commits, because `DagEdge` records node identifiers and no port information (`dag-runtime/src/lib.rs:636-639`). Theorem 5.10(1) is the same fact seen from this side: since `validate` never reads `DagNode::inputs`, there is nothing at the manifest level for a port-preserving map to preserve.

At the harness level, [12] takes a manifest as the wiring component G of an Architecture triple. Theorem 5.2 and Theorem 5.4 establish a finite acyclic schedule, but a single operadic operation additionally requires the root and supplier data of Theorem 3.11. The import is also not sound for a typed reading of G ’s ports, by Theorem 5.10, and Part IV’s use should be read with that restriction in place. Two further restrictions are recorded on the Part IV side and are consistent with the evidence here. A manifest edge is a `OneOrMany` pair on each end (`dag-runtime/src/lib.rs:636-646`), so a manifest is not a tree wiring in Part III’s sense, and the confinement argument Part III proves by induction over a tree does not transfer to it. The operadic reading also splits each multi-output invocation into coordinate generators, so a harness-level result that depends on invocation identity needs the PROP reading or an explicit coordination constraint.

8.4 The policy envelope

`DagToolPolicy` (`dag-runtime/src/lib.rs:566-579`) attaches to each tool a budget in units, an approval flag, and network, filesystem, subprocess and credential policies. What matters here is structural and narrow. The policy is attached per tool, that is per operation, rather than per manifest, and `derived_concurrency_class` reads one field of it, the filesystem write list, when it assigns an effect grade (`dag-executor/src/lib.rs:5406-5425`). Part IV reads that per-operation envelope as a stage-indexed fragment of `Know`, and we adopt that reading by citation, defining no certificate notion of our own. The policy layer is otherwise outside the scope of this paper and is treated in [12].

8.5 Layer dependencies

The dependencies among the four accounts are citations, not definitional entanglements. This paper uses one colour from [10], the port-type alphabet and the $\mathcal{O}$ versus $\mathcal{W}$ question from [11], and the Architecture triple and `Know` from [12]. In the other direction, [11] cites the colour set of Theorem 3.2 in its type-erasure statement, and [12] cites Theorem 3.11 when it reads a manifest as harness-level wiring. No definition here is built on a definition there, or the reverse.

9 Limits and counterexamples

9.1 Open colour set

The prospectus for this series records the objection that `AgentHero`’s `DagIo` ports are string-keyed with JSON payloads rather than drawn from a small closed colour set. Calling the DAG runtime a coloured operad is therefore an idealisation. Theorem 3.10 states the idealisation, and Theorem 3.2 makes it precise: the colour lattice exists, it has a top element, and Theorem 5.10 shows almost every port carries the top. But Theorem 5.11 shows the real cost is elsewhere. Even granting the idealisation entirely, so that every port has a colour and all colours match, L1 still fails, because the runtime performs no check that an input port has a supplier at all. Openness of the colour set is

a modelling inconvenience; absence of a linking check is a structural gap. The two should not be conflated, and Theorem 6.7 shows the repository knows how to close the second.

9.2 Multi-output nodes

An operad represents each output of a multi-output node by a separate generator. A PROP preserves the invocation as one operation, but neither structure makes an external effect deterministic in **Set**. The obstruction is therefore split: output arity is syntactic, nondeterminism is semantic, and shared-store collisions can still violate interchange (Theorem 7.4).

9.3 Post-validation coordination

Code observation 9.1 (Coordination breaks the fixed-composite reading). A manifest declaring `DagCoordination` does not determine a fixed composite. Its worker template nodes are excluded from the executed graph by construction, and the operations that run in their place are created after validation.

Source inspection. `DagCoordination` (`dag-runtime/src/lib.rs:406-420`) declares a mode, a coordinator node, bounds `max_members`, `max_tasks` and `max_depth`, and a closed list of `worker_roles`. `adaptive_worker_template_node_ids` (`dag-runtime/src/lib.rs:1085-1134`) selects, for each declared worker role, the unique node with that role, a kind among `Agent`, `Synthesizer` and `Verify`, and a distributed or either placement carrying the role’s required capabilities. It returns `InvalidCoordinationPolicy` if the candidate is not unique, if the candidate is connected by any edge, or if it is the coordinator. The doc comment records the intent: templates “are intentionally disconnected from the static graph and therefore never execute as ordinary DAG nodes”. `execution_layers` then filters them out of `remaining` before computing the layering (lines 1038, 1046 and 1055). So the manifest’s executed composite omits them, and the tasks the coordinator proposes at run time are not nodes of any composite fixed by Theorem 3.11. $\square$

This is a genuine break rather than a gap. An operad composite is a finite tree fixed in advance. A coordinator that proposes bounded work after validation is a different kind of object. The bounds are static (`max_members` at most 64, `max_tasks` at most 4096, `max_depth` required to equal 1, all checked at `dag-runtime/src/lib.rs:909-969`), but the composite is not. The honest statement is that the operad reading applies to static manifests and that manifests declaring coordination fall outside it.

9.4 Effectful operations

Theorem 5.18 already rules out an algebra of $\mathcal{O}$ realising the executor, on order-sensitivity grounds alone. Effects give a second, independent reason. A node of derived class `ExternalEffect` performs a model call, a process spawn or a network request (`dag-executor/src/lib.rs:5406-5444`). Two invocations with equal inputs need not return equal results, whereas Theorem 2.4 requires each operation to denote a function. So $\mathcal{O}$ as defined here is a syntactic operad and this paper claims no algebra for it.

What is supplied instead is the weaker semantics of Theorem 5.8. The effect word of a run has a well-defined trace, so the runtime does determine an element of a monoid even though it does not determine an element of a store set. A full semantics would need more. The natural target is an algebra valued in the Kleisli category of an effect monad, graded by the classes of Theorem 3.13 in the sense of a parameterised effect monad [8]. In such an algebra the grading of a composite is

computed from the gradings of its parts. The obstruction to constructing it is not the effects but Theorem 5.16: a graded algebra still assigns one value to one operation, so the last-write-wins merge would have to be replaced first. That ordering of the work is the point of this subsection, and it is why the graded algebra is not constructed here.

9.5 Unenforced colour declarations

Example 9.2 (Declared versus enforced colours in `agent-os`). The secondary repository `agent-os` at commit 61cb399 contains a tool registry whose tiers are a natural candidate for a small closed colour set. The module declares `@typetool_tier::builtin|sandbox|external` (`src/tool_interface/lib/tool_interface/registry.ex`, line 35). The registration path for tools discovered from external servers writes `tier::mcp` (line 147), and the listing callbacks match on `:builtin`, `:sandbox` and `:mcp` (lines 177, 181, 185). The declared colour `:external` is never produced, and the produced colour `:mcp` is not declared. Since the host language is dynamically typed, nothing rejects the mismatch.

Theorem 9.2 is the general lesson of this paper in miniature, on a different system by the same author: writing down a finite colour set does not make it a colour discipline. What makes a colour discipline is a check that runs and can fail. `AgentHero` has such a check for plugin capabilities (Theorem 6.7) and for artifact outputs (Theorem 5.10(2)), and does not have one for value ports across edges.

9.6 Threats to the reading

All verdicts are relative to one commit; a validation clause added later would change Theorem 5.10 and Theorem 5.16. The laws L1 to L7 are our formulation, chosen for decidability by inspection; a different but equally faithful axiomatisation could redistribute the verdicts. Theorem 3.10 is doing real work: without it there is no colour alphabet and the question does not arise. A reader who rejects Theorem 3.10 should read this paper as establishing that the operad question is not yet well posed for this runtime, which is a weaker but still substantive conclusion.

The most serious threat, for a reader who wants to check the work independently, is that the primary repository is private. Every code observation names a path, an identifier and a line range at a fixed commit. The source that carries the load-bearing code observations is quoted in the listings, with elisions stated in the captions, so the step from quoted source text to claim can be checked from this document. The step from the repository to the quoted text cannot be, by anyone without access. Nothing here is machine checked, and no code observation is a claim about the behaviour of a running system. An openly archived artifact would be strictly better, and its absence is a limitation of the evidence rather than of the model.

10 Related work

Operads and wiring diagrams. Spivak’s operad of wiring diagrams [21] and its temporal, directed refinement [20] give the standard account of boxes and wires as an operad; Vagner, Spivak and Lerman [22] construct the algebra of open dynamical systems over it. Yau [26] treats coloured operads of wiring diagrams systematically. Leinster [9] and Fong and Spivak [5] are the general references we use for Theorem 2.1. For many-in, many-out operations, Vallette [23] and Markl [16] are the standard sources on properads and PROPs, and Section 7.1 uses their setting. Hasegawa [6] is the reference for traces obtained as least fixed points in a cpo-enriched setting, which is the structure Theorem 6.5 finds missing.

Composition of agent skills. Banu [2] states the correspondence this paper tests and supplies the three-operation vocabulary. That evidence is Banu’s own reference implementation, and the identifiers it names do not occur in the systems studied here. The atomic-skill decomposition that motivates a compositional treatment is due to [15], a preprint withdrawn by its authors after submission; we note it because the correspondence under test relies on it, and we rely on it nowhere. Liu [14] gives a typed lambda calculus for agent composition and reports widespread structural incompleteness in deployed frameworks; Theorem 5.10(1) is an instance of exactly the incompleteness that calculus detects. Wang, Wang and Xu [24] benchmark utility and security of agent skills. That work is cited in [2] under a different author attribution, and we cite the authors listed on the record.

Harness surveys and framing. Meng et al. [18] give an enumerative taxonomy of harness components, against which the categorical account is a contrast rather than a competitor. Pan et al. [25] argue that harnesses are portable objects with algebraic structure; the same work is attributed to a different first author in [2], and we cite the authors listed on the record. Cao [3] supplies the complexity-scaling motivation for externalization that the series’ synthesis takes up. Zhou et al. [27] define the four pillars.

Interfaces and effects. Mazurkiewicz [17] is the source of the independence relation used in Theorem 3.13; Katsumata [8] is the reference for graded effects, which is where a semantics for Section 9’s effectful operations would have to live. Joyal, Street and Verity [7] define the trace that Theorem 6.4 shows **DagLoop** is not.

Prior work by the same author. **agent-os** [13] contains a tool interface layer paper describing a three-tier registry, capability tokens and sandboxed execution in Elixir and OTP. It is a prior instance of the same design problem in a different runtime, and Theorem 9.2 draws its colour-discipline example from its source. That document carries its own “Part II” numbering, which is unrelated to this series’ numbering.

Standards. The Agent Skills manifest format [1] and the Model Context Protocol [19] are the industrial expressions of the pillar; both fix a per-skill interface and neither specifies a composition law, which is the gap a formal account is meant to fill.

11 Conclusion

At commit `1c3ad24`, **AgentHero** validates a finite acyclic dependency order and checks interference before concurrent scheduling. It does not establish input closure, represent a generic identity node, or confine returned value keys to declarations. Consequently the executor does not factor through the proposed operad on all validated manifests: distinct declaration orders can produce distinct stores after a key collision.

The operadic reading is defensible only after additional presentation choices and runtime conditions are stated. A rooted output and unique supplier map determine an operation; actual output disjointness gives order-independent merging; and fully concurrent layers determine one Mazurkiewicz trace class. A PROP preserves multi-output invocation identity but does not remove nondeterminism or shared-store collisions.

12 Open problems

1. Declare the initial-store interface and require every node input to occur there or have a unique compatible ancestor supplier. Determine whether existing manifests satisfy that rule.
2. Confine returned value keys to node declarations, then reject layer-wise sibling declaration

collisions. Together these checks would make declared disjointness sufficient for L4.

3. Extend Theorem 7.4 from the two-strand case to arbitrary layer widths and to horizontal compositions with shared inputs, which a PROP allows and the two-strand statement does not cover. Determine necessary and sufficient conditions when layers share inputs.
4. Order `DagIo` by the pointwise extension of the flat order on JSON values and decide whether, for handlers monotone and continuous in that order, a loop node returns the `max_rounds`-th approximant of a least fixed point. If it does, Theorem 6.4 would become a statement about the absence of a monotonicity requirement on handlers rather than about the loop construct, which would be a more useful diagnosis.
5. Give a semantics for the effectful fragment as a graded algebra over $\mathcal{O}$ using the classes in Theorem 3.13, and determine whether `derived_concurrency_class` is sound for that grading. Soundness here means that a node’s derived class over-approximates the effects its handler can perform.
6. Decide whether the operations proposed at run time by `DagCoordination` can be brought inside a composite by passing to an operad of bounded-depth trees with a schematic node, or whether Theorem 9.1 is a permanent boundary of the reading.
7. Determine whether the plugin capability lattice of Theorem 6.7 and the port colour lattice of Theorem 3.2 can be unified, so that one linking check serves both granularities.

A Code evidence

Every code-backed claim in the paper has a row. Repository `AgentHero` is at commit `1c3ad24`, `agent-os` at `61cb399`. Line numbers are those of the file at the stated commit. Each entry in the final column is an implementation-level statement established by reading the cited lines. The line ranges let a reader with repository access check each citation against the file. The source carrying the load-bearing claims is also quoted in the listings, with elisions stated in the captions, so the step from source text to claim is checkable without access.

Claim	Repo., commit	Path and lines	Identifier	What the code shows
Ports are string-keyed and JSON-valued (Theorem 3.1, Theorem 5.11)	<code>AgentHero</code> <code>1c3ad24</code>	<code>crates/dag-executor/ src/lib.rs</code> 75–83	<code>structDagIo</code>	The implementation at <code>1c3ad24</code> stores ports in two <code>BTreeMaps</code> keyed by <code>String</code> , one with <code>serde_json::Value</code> payloads and one with <code>ArtifactRef</code> payloads
Merge is last-write-wins (Theorem 5.16)	<code>AgentHero</code> <code>1c3ad24</code>	<code>crates/dag-executor/ src/lib.rs</code> 85–90	<code>DagIo::merge</code>	The implementation calls <code>BTreeMap::extend</code> on both components, so a duplicate key takes the argument’s value
The operation signature (Theorem 3.7)	<code>AgentHero</code> <code>1c3ad24</code>	<code>crates/dag-executor/ src/lib.rs</code> 284–292	<code>traitNodeHandler</code>	One asynchronous method from <code>NodeExecutionContext</code> to <code>NodeExecutionResult</code>

Claim	Repo., commit	Path and lines	Identifier	What the code shows
Operation input environment (Theorem 3.6)	AgentHero 1c3ad24	crates/dag-executor/ src/lib.rs 93–108	structNodeExecution Context	Carries manifest, node, a reference to the store, attempt, cancellation token, optional durability receipt and execution world
Operation output (Theorem 3.6)	AgentHero 1c3ad24	crates/dag-executor/ src/lib.rs 160–181	structNodeExecution Result	Carries status, outputs : DagIo , diagnostics, warning, error, command, exit status, model, prompt hash and trace
Operations closed under routing (Section 4)	AgentHero 1c3ad24	crates/dag-executor/ src/lib.rs 313–362	PlacementRouter<L,R >	A NodeHandler built from two handlers, dispatching on the node’s placement mode
Per-app operation family (Table 2)	AgentHero 1c3ad24	crates/dag-executor/ src/lib.rs 366–377	traitDagApp	Extends NodeHandler with dag_type , manifest_file and app_name
A generic runner disclaims domain interpretation (Theorem 5.21)	AgentHero 1c3ad24	crates/dag-executor/ src/lib.rs 379–392	GenericToolRunner	The doc comment states that the runner knows how to invoke tools and record artifacts but not how to interpret domain-specific results; the lines carry no behaviour and establish nothing about identity
Validation precedes execution (Theorem 3.15)	AgentHero 1c3ad24	crates/dag-executor/ src/lib.rs 1601–1613	execute_with_checkpoint	Calls manifest.validate() ? before computing the manifest hash or any layer
Layer siblings share one frozen store (Theorem 5.16)	AgentHero 1c3ad24	crates/dag-executor/ src/lib.rs 1712–1726	layer dispatch	Takes letsnapshot=outputs.clone() once per layer and passes it to every node of the layer
Outputs merged in outcome-vector order (Theorem 5.16)	AgentHero 1c3ad24	crates/dag-executor/ src/lib.rs 1730–1755	outcome loop	Iterates the outcome vector with foroutcomeinoutcomes at line 1730 and calls outputs.merge(produced) at line 1751 on each final successful attempt; that the vector is in declaration order is shown by the row for execute_ordinary_nodes_concurrently
Sequential fallback exists (Theorem 5.7)	AgentHero 1c3ad24	crates/dag-executor/ src/lib.rs 1804–2190	sequential layer path	Executes layer nodes one at a time, cloning the store per node and merging before the next

Claim	Repo., commit	Path and lines	Identifier	What the code shows
Loop is a bounded unrolling (Theorem 6.4)	AgentHero 1c3ad24	crates/dag-executor/ src/lib.rs 2415–2652	execute_loop_node	Initialises visible inputs from the snapshot, dispatches once per round, merges each round's output, and exits early when <code>continue_key</code> is not boolean true
Concurrent dispatch is index-ordered (Theorem 5.16)	AgentHero 1c3ad24	crates/dag-executor/ src/lib.rs 2350–2400	execute_ordinary_node_concurrently	Spawns one task per node tagged with its index and reassembles outcomes into an index-ordered vector
Map is bounded by the manifest (Theorem 6.3)	AgentHero 1c3ad24	crates/dag-executor/ src/lib.rs 2654–2720	execute_map_node	Reads the item list under <code>items_key</code> and, when its length exceeds <code>max_items</code> , returns early with status <code>Failed</code> for a required node and <code>Degraded</code> otherwise
Input schema check is tool-scoped (Theorem 5.10)	AgentHero 1c3ad24	crates/dag-executor/ src/lib.rs 4309–4321 with 4355–4365	validate_node_input_contract, node_tool_schema	The check is guarded by <code>node_tool_schema</code> , which returns <code>None</code> when the node names no tool or the tool declares no <code>input_schema</code> , so the check is skipped in either case
Output schema check is tool-scoped (Theorem 5.10)	AgentHero 1c3ad24	crates/dag-executor/ src/lib.rs 4323–4334 with 4355–4365	validate_node_output_contract	Structured identically to the input check and guarded by the same helper, on <code>output_schema</code>
Artifact outputs are contained in declarations (Theorem 5.10)	AgentHero 1c3ad24	crates/dag-executor/ src/lib.rs 4336–4353, invoked at 3793–3797	validate_node_artifact_output_contract	Fails on an unsafe artifact key or on any artifact key not in <code>node.outputs</code> and ranges over artifacts only; the dispatch path guards the call at lines 3793–3795 on the normalised status being <code>Ok</code> or <code>Degraded</code> and invokes it at line 3797
Empty inputs admits the whole store (Theorem 5.14)	AgentHero 1c3ad24	crates/dag-executor/ src/lib.rs 4367–4405	contract_input_io	Returns <code>inputs.clone()</code> unrestricted when <code>node.inputs</code> is empty, and otherwise filters to the declared names plus loop or map index keys

Claim	Repo., commit	Path and lines	Identifier	What the code shows
The schema fragment (Theorem 3.2)	AgentHero 1c3ad24	crates/dag-executor/ src/lib.rs 4422–4633	validate_contract_s chema	Implements twelve keywords and seven type names and rejects any other declared type name
Composite identity is content addressed (Table 2)	AgentHero 1c3ad24	crates/dag-executor/ src/lib.rs 4652–4664, called at 1609	manifest_hash	Serialises its manifest argument to JSON and returns an FNV-1a 64 digest; the executor calls it at line 1609, immediately after <code>manifest.validate()</code> ? at line 1607
Concurrency requires pairwise compatibility (Theorem 5.6)	AgentHero 1c3ad24	crates/dag-executor/ src/lib.rs 5333–5358	can_execute_layer_c oncurrently	Requires every unordered pair of layer nodes to satisfy <code>node_concurrency_compatible</code>
Grades are derived when undeclared (Theorem 3.13)	AgentHero 1c3ad24	crates/dag-executor/ src/lib.rs 5367–5444	resolved_node_concu rrency, derived_conc urrency_class	The resolver returns a node’s declared <code>DagNodeConcurrency</code> when present and otherwise calls the derivation, which computes a class from the node kind and, for tool nodes, from the tool’s filesystem write policy and executor kind
The independence relation (Theorem 3.13)	AgentHero 1c3ad24	crates/dag-executor/ src/lib.rs 5485–5502	node_concurrency_co mpatible	False if either class is <code>Exclusive</code> , true if either is <code>Pure</code> or both are <code>ReadOnly</code> , otherwise the negation of resource overlap
Five kinds are executor-handled (Section 6)	AgentHero 1c3ad24	crates/dag-executor/ src/lib.rs 6340–6349	is_ordinary_handler _node	Returns false exactly for <code>Branch</code> , <code>Gate</code> , <code>Approval</code> , <code>Loop</code> and <code>Map</code>
<code>DagCall</code> is not expanded by the executor (Section 6)	AgentHero 1c3ad24	crates/dag-executor/ src/lib.rs 5439, 6732	<code>DagNodeKind::DagCall</code>	The only two occurrences assign a concurrency class and a report label; no expansion path exists in this crate
The kind vocabulary is closed (Listing 3)	AgentHero 1c3ad24	crates/dag-runtime/src/ lib.rs 107–125	<code>enumDagNodeKind</code>	Fifteen variants with a <code>serde</code> snake-case representation
Gate policy (Theorem 6.1)	AgentHero 1c3ad24	crates/dag-runtime/src/ lib.rs 180–186	<code>structDagGate</code>	Optional <code>min_usable</code> and a list of source node identifiers
Loop policy (Theorem 6.4)	AgentHero 1c3ad24	crates/dag-runtime/src/ lib.rs 188–193	<code>structDagLoop</code>	A mandatory <code>max_rounds</code> and a <code>continue_key</code> defaulting to <code>loop_continue</code>

Claim	Repo., commit	Path and lines	Identifier	What the code shows
Branch policy (Theorem 6.2)	AgentHero 1c3ad24	crates/dag-runtime/src/ lib.rs 199–206	structDagBranch	A decision key, a map from case label to target node list, and a default target list
Map policy (Theorem 6.3)	AgentHero 1c3ad24	crates/dag-runtime/src/ lib.rs 232–244	structDagMap	Item, index and identifier keys, a mandatory <code>max_items</code> and an optional <code>max_concurrency</code>
Approval policy (Theorem 6.1)	AgentHero 1c3ad24	crates/dag-runtime/src/ lib.rs 254–262	structDagApproval	An approved key and an optional run-level grant key
Effect grades are a closed set (Theorem 3.13)	AgentHero 1c3ad24	crates/dag-runtime/src/ lib.rs 297–308	enumDagConcurrencyClass	Five variants: pure, read-only, workspace mutation, external effect, exclusive
Node inputs are declared but optional (Theorem 5.14)	AgentHero 1c3ad24	crates/dag-runtime/src/ lib.rs 334–373	structDagNode	<code>inputs</code> and <code>outputs</code> are <code>Vec<String></code> with <code>serde</code> defaults
Coordination is bounded and declared (Theorem 9.1)	AgentHero 1c3ad24	crates/dag-runtime/src/ lib.rs 406–420	structDagCoordination	A mode, coordinator node, three numeric bounds and a closed worker-role list
Tools may declare schemas (Theorem 3.2)	AgentHero 1c3ad24	crates/dag-runtime/src/ lib.rs 492–507	structDagTool	Optional <code>input_schema</code> and <code>output_schema</code> as YAML values
Per-operation policy envelope (Section 8)	AgentHero 1c3ad24	crates/dag-runtime/src/ lib.rs 566–579	structDagToolPolicy	Budget units, approval flag, and network, filesystem, subprocess and credential policies
Edges carry node identifiers only (Section 8)	AgentHero 1c3ad24	crates/dag-runtime/src/ lib.rs 636–646	structDagEdge, enumOneOrMany	Endpoints are one or many node identifiers; no port information is recorded
The manifest type (Theorem 3.11)	AgentHero 1c3ad24	crates/dag-runtime/src/ lib.rs 658–676	structDagManifest	Identifier, version, accepts, concurrency, optional coordination, tools, roles, nodes and edges
Validation clauses (Theorem 5.1)	AgentHero 1c3ad24	crates/dag-runtime/src/ lib.rs 684–1025	DagManifest::validate	Checks uniqueness, reference resolution, kind policies, placement, concurrency resources, coordination, parent acyclicity, and ends by computing the layering

Claim	Repo., commit	Path and lines	Identifier	What the code shows
Outputs, not inputs, are name-checked (Theorem 5.10)	AgentHero 1c3ad24	crates/dag-runtime/src/ lib.rs 684–1025, output loop at 810–818	validate	The output loop iterates <code>node.outputs</code> and applies <code>is_safe_artifact_key</code> ; the token <code>inputs</code> occurs in this file only at lines 130, 346, 1087 and 1189, none of which lies inside the body of <code>validate</code>
Coordination bounds are checked (Theorem 9.1)	AgentHero 1c3ad24	crates/dag-runtime/src/ lib.rs 909–969	coordination clause in validate	Requires at most 64 members, at most 4096 tasks, depth exactly 1, an agent or synthesizer coordinator carrying the <code>team.coordinate</code> capability, and worker roles carrying <code>team.work</code>
Parent acyclicity (Theorem 5.1)	AgentHero 1c3ad24	crates/dag-runtime/src/ lib.rs 1004–1022	ancestry walk in validate	Walks the <code>parent_node</code> chain per node and returns <code>ParentNodeCycle</code> on repetition
Layering and cycle detection (Theorem 5.3)	AgentHero 1c3ad24	crates/dag-runtime/src/ lib.rs 1027–1083	execution_layers	Builds a predecessor map from edges alone, peels nodes with no remaining predecessors sorted by declaration index, and returns <code>DagError::Cycle</code> when no node is ready
Worker templates are disconnected (Theorem 9.1)	AgentHero 1c3ad24	crates/dag-runtime/src/ lib.rs 1085–1134	adaptive_worker_template_node_ids	Selects one template per worker role, rejects a template that is connected by any edge, and documents that templates never execute as ordinary DAG nodes
Capability requirements are typed (Theorem 6.7)	AgentHero 1c3ad24	crates/plugin-runtime/ src/lib.rs 480–509	CapabilityRequirement, PluginInstanceSpec	A specification declares required capabilities with an optional flag and provided capability keys
Capability keys are lexically constrained (Theorem 6.7)	AgentHero 1c3ad24	crates/plugin-runtime/ src/lib.rs 511–563	PluginInstanceSpec:: <code>validate</code> , <code>valid_capability_key</code>	Requires a leading lowercase letter and a restricted character set, and rejects duplicate keys on either side

Claim	Repo., commit	Path and lines	Identifier	What the code shows
Linking is enforced (Theorem 6.7)	AgentHero 1c3ad24	crates/plugin-runtime/ src/lib.rs 1126–1164, used at 939–956	resolve_bindings	Binds each requirement to a deterministically chosen active provider and returns <code>None</code> when a non-optional requirement has none; at the activation site a <code>None</code> makes the candidate be skipped, so it never enters the activating state
Manifest read path (Table 2)	AgentHero 1c3ad24	crates/app-sdk/src/lib. rs 511–526	load_dag_manifest	Loads a manifest by DAG type from an app root and fails when the parsed identifier differs from the requested type
A two-node composite (Listing 4)	AgentHero 1c3ad24	agenthero/apps/ platform-smoke/dags/ tool-policy-smoke.yaml 20–33	nodes and edges	A producer declaring <code>outputs:[result.json]</code> , a consumer declaring <code>inputs:[result.json]</code> , and one edge between them
A bounded loop node (Listing 7)	AgentHero 1c3ad24	agenthero/apps/ grokxriv/dags/ review-loop.yaml 158–175	lean_review_fix_code	A loop node with <code>max_rounds:3</code> , two declared inputs and eight declared outputs
A multi-output node (Theorem 7.7)	AgentHero 1c3ad24	agenthero/apps/ grokxriv/dags/ review-loop.yaml 166–174	outputs list	Eight declared output artifact keys on one node
Declared colours are not enforced (Theorem 9.2)	agent-os 6 1cb399	src/tool_interface/lib/ tool_interface/registry. ex 35, 147, 177–187	@typetool_tier, tier	The type alias declares <code>:external</code> ; the registration path writes <code>:mcp</code> and the listing callbacks match <code>:mcp</code>

References

- [1] Anthropic. Agent Skills: an open standard for agent capabilities. <https://agentskills.io>. Announced 16 October 2025; opened as a standard 18 December 2025.
- [2] B. Banu. Harness engineering as categorical architecture. arXiv:2605.12239, 2026.
- [3] Z. Cao. Agentic software: how AI agents are restructuring the software paradigm. arXiv:2606.05608, 2026.
- [4] 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.
- [5] B. Fong and D. I. Spivak. Seven sketches in compositionality: an invitation to applied category theory. arXiv:1803.05316, 2018.

- [6] M. Hasegawa. Recursion from cyclic sharing: traced monoidal categories and models of cyclic lambda calculi. In *Typed Lambda Calculi and Applications (TLCA '97)*, Lecture Notes in Computer Science 1210, pages 196–213, Springer, 1997. DOI 10.1007/3-540-62688-3_37.
- [7] A. Joyal, R. Street, and D. Verity. Traced monoidal categories. *Mathematical Proceedings of the Cambridge Philosophical Society*, 119(3):447–468, 1996. DOI 10.1017/S0305004100074338.
- [8] S. Katsumata. Parametric effect monads and semantics of effect systems. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*, pages 633–646, 2014. DOI 10.1145/2535838.2535846.
- [9] T. Leinster. Higher operads, higher categories. London Mathematical Society Lecture Note Series 298, Cambridge University Press, 2004. arXiv:math/0305049.
- [10] M. Long. Coalgebraic Memory. Part I of the Agentic Engineering series, The YonedaAI Collaboration, YonedaAI Research Collective, 2026.
- [11] M. Long. Typed Protocol Wiring. Part III of the Agentic Engineering series, The YonedaAI Collaboration, YonedaAI Research Collective, 2026.
- [12] M. Long. Harness Architecture. Part IV of the Agentic Engineering series, The YonedaAI Collaboration, YonedaAI Research Collective, 2026.
- [13] M. Long. The AI operating system, part II: tool interface layer, composition, capabilities, and the Elixir/OTP reference implementation. Manuscript, `agent-os` repository, `papers/latex/tool-interface.tex`, commit 61cb399, 2026. The numbering of that document is internal to `agent-os` and unrelated to the present series.
- [14] Q. Liu. λ_A : a typed lambda calculus for LLM agent composition. arXiv:2604.11767, 2026.
- [15] Y. Ma, Y. Liu, X. Yang, et al. Scaling coding agents via atomic skills. arXiv:2604.05013, 2026. Withdrawn by the authors, who report that errors in the data affect the validity of the results. Cited here only to identify a claim made elsewhere; no argument in this paper depends on it.
- [16] M. Markl. Operads and PROPs. In *Handbook of Algebra*, volume 5, Elsevier, 2008. arXiv:math/0601129.
- [17] A. Mazurkiewicz. Trace theory. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, Lecture Notes in Computer Science 255, pages 279–324, Springer, 1987. DOI 10.1007/3-540-17906-2_30.
- [18] Q. Meng, Y. Wang, L. Chen, et al. Agent harness for large language model agents: a survey. Preprints.org 202604.0428.v2, 2026. DOI 10.20944/preprints202604.0428.v2.
- [19] Model Context Protocol contributors. Model Context Protocol specification. <https://modelcontextprotocol.io/specification/2025-06-18>, 2025.
- [20] D. Rupel and D. I. Spivak. The operad of temporal wiring diagrams: formalizing a graphical language for discrete-time processes. arXiv:1307.6894, 2013.
- [21] D. I. Spivak. The operad of wiring diagrams: formalizing a graphical language for databases, recursion, and plug-and-play circuits. arXiv:1305.0297, 2013.

- [22] 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.
- [23] B. Vallette. A Koszul duality for props. *Transactions of the American Mathematical Society*, 359:4865–4943, 2007. arXiv:math/0411542.
- [24] L. Wang, Z. Wang, and A. Xu. SkillTester: benchmarking utility and security of agent skills. arXiv:2603.28815, 2026.
- [25] L. Pan, L. Zou, S. Guo, J. Ni, and H.-T. Zheng. Natural-language agent harnesses. arXiv:2603.25723, 2026.
- [26] D. Yau. Operads of wiring diagrams. arXiv:1512.01602, 2015.
- [27] 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.