

# Harness Engineering as Categorical Architecture

Structural Guarantees Are Harness-Level Properties

Preprint – Feedback Welcome

Bogdan Banu  
bogdan@banu.be

April 2026

## Abstract

The agent harness—the system layer comprising prompts, tools, memory, and orchestration logic that surrounds the model—has emerged as the central engineering abstraction for LLM-based agents. Yet harness design remains ad hoc, with no formal theory governing composition, preservation of properties under compilation, or systematic comparison across frameworks. We show that the categorical Architecture triple  $(G, \text{Know}, \Phi)$  from the ArchAgents framework provides exactly this formalization. The four pillars of agent externalization (Memory, Skills, Protocols, Harness Engineering) map onto the triple’s components: Memory as coalgebraic state, Skills as operad-composed objects, Protocols as syntactic wiring  $G$ , and the full Harness as the Architecture itself. Structural guarantees—integrity gates, quality-based escalation, supported convergence checks—are Know-level certificates whose preservation is structural replay: our compiler checks identity and verifier replay, not output-layer correctness or model behavior. We validate this correspondence with a reference implementation featuring compiler functors targeting Swarms, DeerFlow, Ralph, Scion, and LangGraph: the four configuration compilers preserve three named certificate types by identity/replay, and LangGraph preserves the same certificates through its shared per-stage execution path. The LangGraph compiler creates one node per stage using the same per-stage method as the native runtime, providing LangGraph-native observability without reimplementing harness logic. An end-to-end escalation experiment with real LLM agents confirms that the quality-based escalation control path is model-parametric in this two-model, one-task experiment. The result positions categorical architecture as the formal theory behind harness engineering.

## 1 Introduction

“If you’re not the model, you’re the harness.” This observation, now widespread in the LangChain ecosystem, captures a shift in how practitioners think about LLM-based agent systems. The model provides intelligence; the *harness*—prompts, tool selection, memory management, orchestration logic, safety checks—makes that intelligence actionable. Increasingly, the harness determines whether an agent system is reliable, not the model.

Yet harness engineering remains ad hoc. Practitioners build harnesses by trial and error, guided by blog posts and example repositories rather than formal theory. There is no standard way to state what properties a harness guarantees, whether those properties survive when the harness is compiled to a different framework, or how to compose harnesses without losing guarantees.

We argue that a formal theory already exists, but under a different name. De los Riscos, Corbacho, and Arbib [2] introduce the **ArchAgents** category, where objects are Architecture triples  $(G, \text{Know}, \Phi)$  encoding syntactic wiring, structural knowledge, and deployment maps. Morphisms

are structure-preserving translations. Agents are monoidal functors interpreting architectures in concrete systems. Separately, Zhou et al. [9] identify four pillars of agent externalization: Memory, Skills, Protocols, and Harness Engineering.

Our contribution is the observation that these two frameworks describe the same thing. The four pillars of externalization map directly onto the Architecture triple:

Externalization Pillar	Categorical Role	Concrete Component
Memory	State in the coalgebra	BiTemporalMemory, RunContext
Skills	Objects composed via operad	SkillStage, PatternTemplate
Protocols	Syntactic wiring $G$	WiringDiagram, typed ports
Harness	Full Architecture $(G, \text{Know}, \Phi)$	SkillOrganism + components

The key insight is that structural guarantees—pre-execution integrity checks, quality-based model escalation, supported convergence checks—are represented as Know certificates, not model-specific behavior. When a compiler maps an Architecture to a different framework, our implementation treats certificate preservation as a replay invariant: theorem identity, parameters, and evidence must survive compilation and verify in the target. We validate this with five compiler functors and a two-model, one-task escalation experiment showing that the quality-based escalation control path is model-parametric.

The rest of this paper is organized as follows. Section 2 reviews the externalization and categorical frameworks. Section 3 develops the formal correspondence. Section 4 specifies the certificate-preservation checks and describes the LangGraph per-stage compiler. Section 5 connects atomic skill composition to the operad. Section 6 presents implementation and experimental validation. Section 7 discusses implications and limitations.

## 2 Background

### 2.1 Agent Externalization

Zhou et al. [9] survey the shift from weights-based to harness-centric agent design. They identify four pillars of *externalization*—the process by which cognitive burdens are moved from model parameters to external infrastructure:

1. **Memory:** State across time. Session memory, vector stores, structured knowledge bases. The harness decides what persists, how it is retrieved, and when it expires.
2. **Skills:** Procedural expertise. Tool definitions, workflow templates, progressive skill loading. Skills encode *what* the agent can do, independent of model capability.
3. **Protocols:** Interaction structure. Message formats, turn-taking rules, sub-agent delegation patterns. Protocols govern *how* agents communicate.
4. **Harness Engineering:** The coordination layer itself. Prompt assembly, context management, compaction strategies, evaluation-driven refinement. The harness is the system that orchestrates memory, skills, and protocols into coherent agent behavior.

Their key observation: “agent infrastructure matters not merely because it adds auxiliary components, but because it transforms hard cognitive burdens into manageable forms.” This transformation is exactly what formalization should capture.

## 2.2 The ArchAgents Category

De los Riscos, Corbacho, and Arbib [2] introduce a category-theoretic framework for comparing agent architectures. An **Architecture** is a triple  $A = (G_A, \text{Know}_A, \Phi_A)$  where:

- $G_A$  is the **syntactic wiring**—a graph of modules, ports, and directed edges describing how information flows.
- $\text{Know}_A$  is the **knowledge structure**—the set of structural properties, invariants, and certificates the architecture maintains.
- $\Phi_A$  is the **deployment map**—the mapping between abstract capability slots and concrete model/tool implementations.

A **morphism**  $f : A \rightarrow B$  is a structure-preserving translation: it maps modules in  $G_A$  to modules in  $G_B$ , knowledge in  $\text{Know}_A$  to knowledge in  $\text{Know}_B$ , and deployment maps in  $\Phi_A$  to deployment maps in  $\Phi_B$ . In practice, a morphism is a *compiler*—a function that transforms one agent architecture into another while preserving structural properties.

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

## 2.3 Alternative Formalizations

Liu [3] independently develops a typed lambda calculus ( $\lambda_A$ ) for agent composition, extending the simply-typed lambda calculus with oracle calls, bounded fixpoints, probabilistic choice, and mutable environments. The work demonstrates that five mainstream frameworks (LangGraph, CrewAI, AutoGen, OpenAI SDK, Dify) embed as typed  $\lambda_A$  fragments, and finds that 94.1% of 835 real-world GitHub agent configurations are structurally incomplete under this formalization. Where  $\lambda_A$  provides type-theoretic guarantees (type safety, termination), our categorical approach provides *property preservation under compilation*—a complementary guarantee.

Willström et al. [8] propose Natural-Language Agent Harnesses (NLAH), externalizing harness behavior as portable, editable natural-language artifacts with explicit contracts. Their Intelligent Harness Runtime (IHR) executes these artifacts through durable contracts and lightweight adapters. This validates a key assumption of our work: that harnesses are portable objects with algebraic structure, not implementation-specific code. Our Architecture triple  $(G, \text{Know}, \Phi)$  provides the categorical formalization that NLAH’s natural-language artifacts implicitly require.

Chen et al. [1] introduce SkillTester, a comparative quality-assurance harness for agent skills that benchmarks both utility and security. Their rubric-based skill evaluation parallels our Verifier-Component, which scores stage outputs against task-specific rubrics and triggers model escalation when quality falls below threshold.

## 2.4 Cross-Domain Corroboration

Marom et al. [5] reach a structurally identical compositional-verification framework while solving a fundamentally different problem: translating biological mechanisms (a hygromorphic pinecone) into 4D-printed engineered systems. They define a category  $\text{Dyn}$  of stimulus-response dynamical systems with subcategories  $\text{Nat} \subset \text{Dyn}$  (natural) and  $\text{Art} \subset \text{Dyn}$  (artificial), an implementation

functor  $\mathcal{F}: \mathbf{Nat} \rightarrow \mathbf{Art}$ , a specification space  $\mathbf{Spec}$  with realization projection  $\pi: \mathbf{Spec} \rightarrow \mathbf{Art}$ , and a compilation functor  $\mathcal{E}: \mathbf{Spec} \rightarrow \mathbf{Comp}$  translating verified specs to machine instructions. Their central theorem is closure of a simulation condition under composition—structurally the same preservation property we specify and check operationally in Section 4 for the certificate-preserving morphisms of the ArchAgents category.

The relevance is not the materials substrate but the principle they articulate explicitly: *“Two systems that share the same compositional structure ...can be related by a functor that preserves the interface logic at every scale without requiring that the two domains share any physical substrate.”* This substrate-independence is the materials-engineering articulation of the framework portability we claim for ArchAgents—a transformer-LLM agent and a Python LangGraph runtime share no more substrate than a cellulose pinecone and a polymer 4D bilayer, yet in both cases a structure-preserving functor relates them by interface logic alone. Their 2×2 generativity demonstration, in which one of four printed designs (thermal twisting) arises purely by composition of components validated for other products, is independent, non-LLM, physically-fabricated evidence for the same compositional claim. We make no equivalence claim: their framework is substrate-validated through fabrication and measurement, ours is property-validated; only the compositional logic and the substrate-independence principle are shared.

## 2.5 Enumerative and Categorical Views

Meng et al. [6] present a contemporaneous engineering taxonomy, formalizing the harness as a six-component tuple  $H = (E, T, C, S, L, V)$  for execution loop, tool registry, context manager, state store, lifecycle hooks, and evaluation interface, and surveying over 110 papers and 23 systems on a Harness Completeness Matrix. Among the nine open challenges they identify, their accompanying repository overview states: *“Formal verification, portability testing, and protocol interoperability all require compositional reasoning that current research has not addressed.”*

This paper addresses that open challenge for one direction—certificate preservation under harness compilation—through the Architecture triple  $(G, \text{Know}, \Phi)$  and the structure-preserving morphisms of Section 4. The  $(E, T, C, S, L, V)$  taxonomy refines the implementation surface of an ArchAgents object:  $E$  and  $T$  contribute wiring and realizations,  $C$  and  $S$  expose state, and  $L$  and  $V$  are the natural surfaces on which Know-level certificates can be defined and replayed. A precise mapping between the two formalizations is developed elsewhere.

## 2.6 The Gap

These frameworks describe the same phenomenon from different angles. Zhou et al. provide the *engineering* vocabulary (Memory, Skills, Protocols, Harness). De los Riscos et al. provide the *mathematical* structure (objects, morphisms, functors). Liu provides *type-theoretic* guarantees. Willström et al. validate *portability*. Meng et al. provide the *enumerative* component taxonomy and explicitly mark compositional verification as open (Section 2.5). None of these strands connect to the others. The result is that harness engineers build without formal guarantees, and theorists formalize without grounding in practice. This paper bridges the gap.

# 3 The Correspondence

We now develop the formal mapping between the externalization pillars and the Architecture triple. The correspondence is not a loose analogy—each pillar maps to a specific categorical component with concrete implementation.

### 3.1 Memory as Coalgebraic State

Agent memory is state that persists across interactions. In the categorical framework, state is modeled as a **coalgebra** for a polynomial functor: a pair  $(S, \phi)$  where  $S$  is the state space and  $\phi : S \rightarrow P(S)$  is the transition function.

In practice, this means memory is not just “stored data” but a dynamical system with update rules. Our implementation uses **bi-temporal memory** [7]: every fact carries both a *valid time* (when it was true in the world) and a *record time* (when the system learned it). This dual-time structure enables belief-state reconstruction: “what did the agent know at decision point  $t$ ?”

The `RunContext` class wraps the shared state dict with typed property accessors, providing the coalgebraic state interface that components read from and write to during execution.

### 3.2 Skills as Operad-Composed Objects

Skills are procedural capabilities. In the categorical framework, they are **objects composed via an operad**—a structure that defines how components can be wired together.

Three composition operations are available:

- **Serial** ( $B \circ A$ ): The output of stage  $A$  feeds into stage  $B$ . Sequential pipelines.
- **Parallel** ( $A \otimes B$ ): Stages  $A$  and  $B$  execute independently with disjoint state. Specialist decomposition.
- **Trace** ( $\text{Tr}(A)$ ): A feedback loop where some outputs feed back as inputs. Homeostatic control.

Ma et al. [4] demonstrate empirically that five atomic coding skills—localize, edit, test, re-produce, review—compose without negative interference under joint training. This is precisely the behavior the operad is designed to support structurally: typed composition can preserve component properties when the relevant property is closed under the chosen operation.

### 3.3 Protocols as Syntactic Wiring $G$

Protocols define how agents communicate. In the Architecture triple, this is  $G$ —the syntactic wiring graph. Each module has typed input and output ports; wires connect outputs to inputs with type compatibility checks.

The wiring diagram is not just a topology—it carries **integrity labels** on each port (validated, raw, sanitized) and supports three optical types at the wire level: lenses (constitutional access), prisms (conditional routing), and traversals (batch processing). These wire-level optics formalize the “interaction structure” that Zhou et al. identify as the Protocols pillar.

### 3.4 Harness as Full Architecture $(G, \text{Know}, \Phi)$

The harness is not one component of the Architecture—it *is* the Architecture. The `Know` component deserves special attention: it encodes the structural guarantees that make the harness more than a bag of parts.

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

Examples of certificates in our implementation:

- **Priority gating:** “Critical operations are served under any load” (from metabolic state machine parameters).
- **No false activation:** “Normal traffic never triggers quorum sensing” (from steady-state dynamics).
- **No oscillation:** “mTOR scaling converges” (from feedback gain bounds).

These certificates are attached to `Know`, not to any specific model. When the model changes—or when the architecture is compiled to a different framework—the supported certificates remain replayable when their hooks and parameters are preserved.

### 3.5 The Deployment Map $\Phi$

The deployment map  $\Phi$  maps abstract capability slots to concrete implementations. In practice, this is the **mode-to-model mapping**: each stage declares a cognitive mode (observational/action-oriented), and the harness resolves this to a specific model tier (fast/deep).

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

This separation is what makes the harness model-parametric. The same harness (same  $G$ , same `Know`) can run on different models by changing  $\Phi$ . The structural certificates in `Know` are replayed against preserved hooks and parameters rather than tied to a specific model assignment.

## 4 Certificate Preservation

The correspondence in Section 3 would be merely descriptive if structural guarantees could not survive compilation. This section specifies the preservation checks used by the implementation.

### 4.1 Compiler Functors

A **compiler functor**  $F : \text{Arch}(\text{Source}) \rightarrow \text{Arch}(\text{Target})$  maps one architecture to another. In our implementation, each compiler produces a serializable configuration dict for a specific orchestration framework:

- `organism_to_swarms`: `Operon`  $\rightarrow$  `Swarms` graph workflow
- `organism_to_deerflow`: `Operon`  $\rightarrow$  `DeerFlow` `LangGraph` session
- `organism_to_ralph`: `Operon`  $\rightarrow$  `Ralph` hat configuration
- `organism_to_scion`: `Operon`  $\rightarrow$  `Scion` grove deployment
- `organism_to_langgraph`: `Operon`  $\rightarrow$  `LangGraph` `StateGraph`

Each compiler is wrapped in a `CompilerFunctor` class that automatically extracts source and target architectures, verifies the functor laws, and produces a `PreservationResult` with three checks:

1. **Graph preservation:** source stage names  $\subseteq$  target stage names, source edges  $\subseteq$  target edges.

2. **Certificate replay invariant:** source certificate theorems  $\subseteq$  target certificate theorems, source parameters/evidence are preserved, and all certificates `verify()`  $\rightarrow$  `holds=True`.
3. **Deployment-map preservation:** source stage names present in target (mode mapping may differ).

## 4.2 The LangGraph Compiler

The LangGraph compiler `organism_to_langgraph()` creates one LangGraph node per organism stage. Each node calls `organism.run_single_stage()`—the same per-stage method that `organism.run()` uses internally. Conditional edges route based on the return value: "continue" proceeds to the next stage, "halt" or "blocked" routes to the graph's terminal node.

This design avoids the reimplementation trap: an earlier attempt that encoded component logic directly as LangGraph nodes required 10 rounds of code review to achieve behavioral parity with `organism.run()`. By extracting `run_single_stage()` from the run loop and calling it from both paths, the LangGraph graph uses the *same code path* as the native runtime—zero reimplementation.

**Implementation invariant (per-stage certificate replay).** The LangGraph compiler preserves certificate replay for organisms whose certificates use the supported component hooks (CertificateGate, VerifierComponent, WatcherComponent). Such certificates verify after compilation because each stage node executes with those hooks intact, and interventions (RETRY, ESCALATE, HALT) are handled within the node before routing.

The per-stage graph also provides LangGraph-native observability: each stage appears as a visible node in LangGraph Studio, enabling inspection of individual stage execution, timing, and intervention decisions. The tested certificates hold because `run_single_stage()` preserves the hooks and parameters they replay, not because they are independent of every possible graph topology.

**Parallel stage groups.** Organisms with parallel stage groups compile to a fork/join topology using LangGraph's native `Send` API. Each parallel group generates three types of infrastructure nodes: a *fork* node that snapshots state and dispatches isolated copies to each stage via `Send`, the individual *stage* nodes (one per parallel stage, calling `run_single_stage()` as usual), and a *join* node that merges results using the same conflict-detecting merge logic as the native runtime. Sequential organisms are unchanged—each stage maps to a single node as before.

Certificate replay holds for the supported certificates in parallel groups because each stage node still calls `run_single_stage()` with full component hooks. The fork/join infrastructure is purely topological—it manages state isolation and result aggregation, not structural guarantees.

## 4.3 Empirical Verification

We verify the implementation invariant across the five compiler targets, with LangGraph checked through per-stage execution rather than the configuration-dict path:

Compiler	Graph	Certificates	Interface
Swarms	✓ (1:1)	✓ (100%)	✓
DeerFlow	× (hub-spoke)	✓ (100%)	✓
Ralph	× (hats)	✓ (100%)	✓
Scion	✓ (+watcher)	✓ (100%)	✓
LangGraph	✓ (per-stage)	✓ (100%)	✓

Across the five compiler targets tested, the three supported certificate types preserve identity and verify. Swarms and LangGraph preserve per-stage structure. DeerFlow reshapes to hub-and-spoke. Ralph converts to hat patterns. Scion adds a watcher agent. In these tests, the structural guarantees survive because they are Know-level replay artifacts whose hooks and parameters are preserved; the tested certificates do not depend on exact graph topology beyond those hooks and parameters.

## 5 Atomic Skills as Operad Composition

Ma et al. [4] identify five atomic coding skills that serve as “basis vectors for complex software engineering tasks”: code localization, code editing, unit-test generation, issue reproduction, and code review. Under joint reinforcement learning, these skills compose without negative interference—improving one does not degrade another—achieving 18.7% average improvement over task-specific optimization.

This result has a categorical interpretation. Each atomic skill is an **object in the agentic operad**: a typed module with input ports, output ports, and a capability annotation. The operad’s composition operations (serial, parallel, trace) define how skills can be combined. Ma et al.’s finding that skills compose without interference is precisely the claim that operad composition **supports property-preserving composition when the relevant certificate is closed under the operation**—the categorical condition corresponding to non-interference.

Our implementation registers the five atomic skills as `PatternTemplate` objects with typed `TaskFingerprints`:

Skill	Shape	Topology	Roles
Localize	sequential	<code>skill_organism</code>	searcher, ranker
Edit	sequential	<code>skill_organism</code>	editor, validator
Test	sequential	<code>skill_organism</code>	analyzer, generator, runner
Reproduce	sequential	<code>skill_organism</code>	reader, executor, verifier
Review	parallel	<code>specialist_swarm</code>	logic, style, security, reporter

The distinction between sequential skills (pipeline topology) and the parallel review skill (swarm topology) is automatically resolved by the shared `_shape_to_topology()` function used across all convergence adapters.

The connection to harness engineering: atomic skills are the  $G$  component of a harness—the wiring that defines how work flows. The certificates attached to a skill-based harness (e.g., “all review axes are covered”) are Know-level properties that survive composition and compilation.

## 6 Implementation and Experiments

### 6.1 Reference Implementation

The correspondence is implemented in the Operon framework (`operon-ai`, MIT license). Key components:

- **Architecture extraction:** `extract_architecture(organism)` produces the  $(G, \text{Know}, \Phi)$  triple from any `SkillOrganism`.
- **Compiler functors:** Five `CompilerFunctor` instances with automatic preservation verification.



- **Certificate framework:** Self-verifying certificates with `certify()` and `verify()` methods on certifiable components (ATP budget, quorum sensing, mTOR scaling).
- **Atomic skills catalog:** `seed_library_from_atomic_skills()` registers the five skills with correct topology mapping.
- **LangGraph compiler:** `organism_to_langgraph()` creates one LangGraph node per stage, each calling `run_single_stage()`.

## 6.2 Escalation Experiment

To test whether structural guarantees are genuinely harness-level, we designed an experiment that changes the model while keeping the harness constant.

**Setup.** A single-stage code review organism with:

- **Fast model:** Phi-3 Mini (3.8B parameters) via Ollama
- **Deep model:** Gemma 4 (27B MoE, 4B active) via Ollama
- **VerifierComponent:** Rubric-based quality evaluation using Gemma 4 as judge
- **WatcherComponent:** Verifier-threshold escalation after quality evaluation
- **Quality threshold:** 0.6 (below this, the watcher fires ESCALATE)

The task is `hard_par_08`: a code review containing four subtle bugs (off-by-one pagination, TOCTOU cache race, float precision in financial math, `except Exception` swallowing `KeyboardInterrupt`).

**Results.**

Phase	Model	Quality	Action
Initial execution	Phi-3 Mini (fast)	0.50	EXECUTE
Verifier evaluation	Gemma 4 (judge)	—	Score = 0.50
Watcher decision	—	$0.50 < 0.60$	ESCALATE
Escalated execution	Gemma 4 (deep)	—	EXECUTE

The escalation fires correctly: Phi-3 Mini produces a review that the judge scores below threshold, the watcher escalates to Gemma 4, and the deep model re-executes the stage. In this two-model, one-task experiment, the quality-based escalation control path is model-parametric: it is represented as harness structure (`Know`) and executed under the chosen deployment map ( $\Phi$ ).

**Discrimination.** Independent validation of the task confirms discrimination between weak and strong models: Phi-3 Mini scores 0.72 (identifies bug classes but not concrete failure scenarios), Gemma 4 scores 1.00 (explains each failure scenario in detail). Quality delta = 0.28, exceeding the 0.20 threshold.

### 6.3 Certificate Preservation Measurement

We compile organisms with multiple certificates (ATP priority gating + quorum sensing no-false-activation + mTOR no-oscillation) across all four non-LangGraph compilers and measure preservation:

Compiler	Certificates Preserved	Verified
Swarms	3/3 (100%)	3/3
DeerFlow	3/3 (100%)	3/3
Ralph	3/3 (100%)	3/3
Scion	3/3 (100%)	3/3

Full certificate identity (theorem + parameters + source) is preserved across all four measured compilers, providing empirical evidence for the operational certificate-replay invariant. The LangGraph compiler preserves certificates via per-stage execution of `run_single_stage()` (Section 4).

### 6.4 SWE-bench-lite: 8B Format Discipline Is the Ceiling

To probe whether the categorical harness delivers end-to-end value beyond certificate preservation, we ran an apples-to-apples comparison on SWE-bench-lite: 10 real Python bug-fix instances from `astropy` and `django`, with patches applied inside the official SWE-bench Docker harness and evaluated via `FAIL_TO_PASS` + `PASS_TO_PASS` test suites. An initial 2026-04-16 run was inconclusive: 28 of 30 submissions failed at `git apply`, mixing model weakness with prompt-format/pathing issues that we could not separate. We therefore built a *patch-apply pipeline* (sanitizer + repository grounding) and reran. With grounding active, the failure mode shifts from `git apply` errors to honest `empty_patch` rejections, and the run *does* support a capability conclusion: at 8B / Q4\_K\_M, the model’s diff-format discipline is below what SWE-bench-lite requires.

**Setup.** Gemma 4 via Ollama (tag `gemma4:latest`, digest `c6eb396dbd59`, blob `sha256:4c27e0f5`, architecture `gemma4`, 8.0B parameters, Q4\_K\_M quantization, 131,072 context). The immutable digest is recorded in `eval/results/swebench_phase2.json`. Three conditions:

- **baseline:** direct single prompt, request unified diff.
- **organism:** three-stage `SkillOrganism` (*localize* → *edit* → *verify*).
- **langgraph:** same organism, compiled via `organism_to_langgraph()` and executed as a LangGraph `StateGraph`.

The pipeline applied to all three conditions in this rerun:

- Each instance’s repository is shallow-cloned at its `base_commit` into a local cache; up to five candidate file snippets selected by issue-text heuristics are injected into the prompt as repository context.
- Model output is sanitized: paths normalized (e.g. stripping the `owner/repo/` prefix the model often hallucinates), hunk headers checked for placeholder line numbers (`@@ -XXX,N +XXX,N @@` style), hunk bodies validated against declared counts, and patches whose paths don’t exist in the cloned tree fuzzy-corrected against unique basename matches or rejected. Patches that the sanitizer cannot make safe are dropped (recorded as `empty_patch`, never forwarded to `git apply`).

## Results (per-condition outcome distribution, 10 instances per condition).

Condition	Resolved	Unresolved	Sanitizer-rejected	Runtime error	Evaluated	Mean latency*
baseline	0	1	7	2	1/10	131 s
organism	0	0	10	0	0/10	170 s
langgraph	0	0	10	0	0/10	172 s

\*Mean latency is computed over predictions that completed (the model returned text); the two baseline runtime errors were API timeouts where the call never returned and have no meaningful latency.

Categories: **resolved** (patch applied, target tests pass), **unresolved** (patch applied, tests fail), **sanitizer-rejected** (the model returned a diff-shaped output but the patch sanitizer dropped it for placeholder hunk headers, malformed counts, or invented paths — harness sees this as **empty\_patch**), **runtime error** (the model call itself raised before returning, e.g. Ollama API timeout). **Evaluated** counts only instances that reached a pass/fail harness verdict (resolved + unresolved). Zero **git apply** errors occurred: sanitizer pre-rejection ensures no malformed patch reaches the harness.

**Comparison with the 2026-04-16 run.** The earlier (un-grounded, un-sanitized) run had baseline 1/10 evaluated with 9 **error**, organism 1/10 with 5 **error** + 4 **empty\_patch**, langgraph 0/10 with 6 **error** + 4 **empty\_patch**. Two changes are notable. First, every **error** category collapsed to zero: the sanitizer refuses to forward patches that **git apply** would reject. Of the 27 model returns that reached the sanitizer in the rerun, 27 were dropped (placeholder hunks, malformed counts, or invented paths); the remaining 3 submissions across conditions either produced one applicable patch (django-11001 baseline) or never returned (the 2 baseline API timeouts on astropy-12907 and astropy-14995 — these are recorded as **runtime\_error** rather than **empty\_patch** so a downstream reader can distinguish "model produced invalid output" from "model never produced output"). Second, the organism/baseline **empty\_patch** gap from the original run (4 vs. 0) closed: the tightened "[**edit**]" stage emits a single fenced diff and nothing else" instruction eliminated the previous multi-stage format leak. The remaining 1-vs-0 gap (baseline produces one applicable diff, organism and LangGraph produce none) is consistent with the localize/edit decomposition asking the model to write a self-contained diff while juggling stage outputs, a discipline tax the single-shot baseline avoids.

**Interpretation: 8B format discipline, not file selection, is the ceiling.** The original run confounded two failure modes: (a) the model selecting nonexistent or wrong file paths, (b) the model writing hunk headers and bodies that **git apply** cannot consume. Grounding reduced (a) but did not eliminate it: every prompt now contains the actual files at **base\_commit**, yet some outputs still name paths that do not exist in the cloned tree. The dominant remaining rejection is (b), hunk-format errors—placeholder hunk headers and mismatched header counts—which is separable from residual path errors. At this model scale, providing the actual files in the context window does *not* translate to format-correct unified diffs. The 2 runtime errors are noise from the local Ollama API and do not refute this conclusion: those instances also failed via organism and langgraph (which completed normally), so the model never produced a usable diff for them either.

The single survivor, **django/django-11001** (baseline), reached the harness as **unresolved**: the patch applied cleanly but the test suite still failed. That same instance was the one survivor in the 2026-04-16 run as well, suggesting a structural property of that issue (likely a small, localized hunk) makes it more amenable to small-model output.

**Latency cost of grounding.** Baseline mean latency (over the 8 completed calls) rose from 44 s in the original run to 131 s with grounding, organism from 88 s to 170 s, langgraph from 90 s to 172 s. The 30 KB of repository context in each prompt roughly doubles to triples per-call wall-clock with no compensating gain in evaluated instances at this model scale. Grounding’s cost-benefit changes with stronger models, but at 8B it is overhead.

**What this rerun does and does not say.** It does say: at 8B / Q4\_K\_M, the model’s ability to emit `git apply`-clean unified diffs is the binding constraint on SWE-bench-lite resolution, and three-stage decomposition does not relax it. It does not say that the organism architecture is intrinsically worse than direct prompting — with 1 vs. 0 evaluated instances, the experiment is still under-powered to discriminate. The harness-level *structural* guarantees reported in Sections 4 and in the escalation experiment are unaffected: they are Know-level properties verified independently of SWE-bench task resolution.

This motivates Section 7: the demonstrated transfer value of categorical harness engineering in this paper is in *preservation* (certificate identity across compilers) and *primitive reuse* (escalation), not in task-resolution gains on SWE-bench-lite at this model scale.

#### 6.4.1 Phase C: Cross-Model Check with Format-Correction Retry

The v0.34.5 result above named two follow-up wedges: a stronger model and a format-correction retry loop. Paper 5’s released code (v0.35.0) implements the retry and runs a cross-model check against a second locally-available 8B model, answering both wedges at once. The aim is not a better SWE-bench score — prior results already place this outside the reach of 8B-class local models — but a test of whether the 8B format-discipline ceiling reported above generalizes across training regimes, and whether a reason-coded retry prompt can recover any of the sanitizer-rejected submissions.

**Setup.** `deepseek-r1:8b` via Ollama (tag `deepseek-r1:8b`, digest `6995872bfe4c`, blob `sha256:e6a7edc1`, architecture `qwen3` — the Ollama weight is the DeepSeek-R1 distillation onto a Qwen3-8B base, 8.2B parameters, Q4\_K\_M quantization, 131,072 context). Same 10 SWE-bench-lite instances as the v0.34.5 run, same grounded prompt pipeline, three conditions (baseline, organism, langgraph). The new flag `--retry-on-reject` is active: when the sanitizer rejects a submission, the runner re-prompts the model once with the specific rejection reason code (one of `placeholder_hunk`, `truncated_hunk`, `overlong_hunk`, `path_not_found`, `ambiguous_path`, `empty_extraction`, `malformed_metadata`), embedded in the retry prompt, followed by the failed output verbatim. Artifact at `eval/results/swebench_phaseC`.

**Results (deepseek-r1 with retry, per-condition, 10 instances each).**

Condition	Resolved	Unresolved	Sanitizer-rejected	Runtime error	Evaluated	Mean latency
baseline	0	0	10	0	0/10	1084 s (18 min)
organism	0	0	10	0	0/10	1073 s (18 min)
langgraph	0	0	10	0	0/10	1130 s (19 min)

Zero submissions reached the harness under any condition. Zero retry-recovered patches across all 30 submissions, even with targeted reason-code prompts. Zero runtime errors (the 900 s timeout introduced in v0.34.5 absorbs the long `<think>` blocks reasoning models produce; the mean per-call latency is 18–19 min vs gemma4’s 2–3 min, but every call completed cleanly).

### Comparison with gemma4 v0.34.5.

	gemma4:latest (v0.34.5)	deepseek-r1:8b (v0.35.0)
Total evaluated	1/30	0/30
Sanitizer-rejected	27	30
Runtime errors	2	0
Resolved	0	0
Retry active	no	yes
Retry-recovered patches	—	0
Mean latency (baseline, completed)	131 s	1084 s

Two observations.

First, **the ceiling is consistent across the two tested local 8B Q4\_K\_M models.** Both models (gemma4 instruction-tuned, deepseek-r1 reasoning-distilled-onto-Qwen3) produce zero resolved instances. Gemma4 crossed the sanitizer once; deepseek-r1 crossed it zero times. The single v0.34.5 “survivor” (django-11001 baseline, `unresolved`) was gemma4-specific: deepseek-r1 also could not produce a `git apply`-clean diff for that instance under any of the three conditions. Within this two-model local setup, the 8B-class format-discipline ceiling reported in v0.34.5 is not a single-model artifact.

Second, **retry-with-reason-code did not break the ceiling**, and the reason distribution itself is informative. The v0.35 artifact records each submission’s final sanitizer reason (per-instance `sanitize_reason` field). For deepseek-r1:8b across 30 submissions: 26 `empty_extraction` (the model’s output contained no diff-shaped content at all), 3 `overlong_hunk` (body content exceeded declared counts), 1 `truncated_hunk` (body shorter than declared counts); zero submissions hit `placeholder_hunk`, `path_not_found`, or `ambiguous_path` (so the grounding-specific reason codes covering file-selection failure didn’t fire even once — consistent with the v0.34.5 claim that file selection is not the bottleneck). The retry mechanism itself is sound: the callback is invoked exactly when expected, the retry prompts embed the reason code and the failed output verbatim with reason-specific guidance (“use real integer line numbers” for `placeholder_hunk`; “use exact paths from the repository context” for `path_not_found`; “respond with a single fenced diff block and nothing else” for `empty_extraction`; etc.), and the artifact records every submission’s `retry_attempted=true` with `retry_recovered=false`. What the retry cannot do is move the model across the capability boundary. At 8B, the dominant failure mode is “doesn’t produce diff-shaped output in the first place” (26/30), not “produces diff-shaped output with fixable errors” (4/30). Retry-with-guidance is calibrated for the second regime but the model sits mostly in the first. This is a sharper negative result than the v0.34.5 paper predicted (we suggested retry “might recover 20–40% of sanitizer drops”); at 8B, it recovers zero.

**What Phase C does and does not say.** It says: (i) the 8B format-discipline ceiling holds across training regimes, (ii) reasoning-distilled training does not, on its own, improve diff-format production, (iii) retry-with-targeted-guidance is the wrong lever for a capability ceiling and requires a model that is at least close to producing correct output — which 8B models largely are not. It does not say retry is universally useless; stronger models that make occasional format mistakes may recover meaningfully. Cloud-GPU reruns against a 70B class model are the natural next wedge, outside this paper’s local-only scope. The patch-apply pipeline (sanitizer + repo grounding + reason-coded retry) is instrumented for exactly that follow-up.

## 7 Discussion and Conclusion

### 7.1 What the Correspondence Provides

The mapping between externalization pillars and the Architecture triple is not a metaphor—it is a formal correspondence with executable implementation and verified properties. Concretely, it provides:

1. **A language for harness design:** Instead of ad hoc descriptions (“add a safety check”), engineers can specify certificates in Know with verifiable semantics.
2. **Preservation guarantees:** When compiling a harness to a different framework, functor-law checks are paired with explicit certificate identity and replay checks. This is mechanically verified for three supported certificate types across five targets.
3. **Composition rules:** The operad defines how skills combine (serial, parallel, trace) with type safety. Ma et al.’s empirical finding that atomic skills compose without interference is consistent with this closure-conditional structural view.
4. **Model parametricity:** Structural guarantees reside in Know, not in a fixed model choice. The escalation experiment demonstrates this concretely for one task and two local models.

### 7.2 Limitations

This work has several honest limitations:

- **Static framework:** The Architecture triple is a snapshot. We do not model how harnesses evolve over time (e.g., learning from experience, adapting to distribution shift). Dynamic harness evolution is an open problem.
- **Single reference implementation:** All validation uses one codebase (Operon). The correspondence needs independent implementations to confirm generality.
- **Certificate scope:** Current certificates cover structural invariants (priority gating, convergence bounds) but not behavioral properties (“the agent never hallucinates”). Extending Know to behavioral certificates is future work.
- **Escalation experiment scale:** The escalation experiment uses two models on one task. Scaling to diverse tasks and model families would strengthen the claim.
- **SWE-bench-lite ceiling at 8B, confirmed cross-model:** Section 6.4 reports 0 resolved instances across baseline, organism, and LangGraph on 10 SWE-bench-lite instances with Gemma 4 (8B / Q4\_K\_M), under a grounded rerun where each prompt contains the relevant files from the target repository at `base_commit` and the model output is sanitized before submission. Section 6.4.1 reruns the same instances against a second locally-available 8B model (`deepseek-r1:8b`, DeepSeek-R1 distilled onto Qwen3-8B, Q4\_K\_M), with a format-correction retry loop active: when the sanitizer rejects, the runner re-prompts once with the specific reason code and the failed output embedded. Result: 0/30 evaluated, zero retry-recovered patches across all 30 submissions. Gemma4’s one sanitizer-crossing instance (`django-11001` baseline, `unresolved`) was gemma4-specific; `deepseek-r1` could not produce a `git apply`-clean diff for it either. The 8B-class format-discipline ceiling is not a single-model artifact, and retry-with-targeted-guidance is

the wrong lever for a capability ceiling (it helps models that occasionally misformat, not models that cannot format at all). Consequently, our demonstrated transfer value in this paper is *preservation* (Section 4) and *primitive reuse* (the escalation experiment); task-resolution gains on SWE-bench-lite require a substantially stronger model, outside this paper’s local-only scope.

### 7.3 Conclusion

Harness engineering is not ad hoc—it has a categorical formalization. The Architecture triple  $(G, \text{Know}, \Phi)$  from the ArchAgents framework is the formal theory behind the harness concept that the LangChain ecosystem has converged on empirically. Structural guarantees are Know-level replay artifacts preserved in the tested compiler suite by explicit identity and verifier-replay checks. Atomic skills compose via the operad under closure conditions. The model is selected by  $\Phi$ ; changing it changes the deployment map, while supported certificates remain tied to preserved hooks and parameters.

The practical implication: before building a harness, define its certificates. Before compiling to a new framework, verify preservation. Before composing skills, check type compatibility. The categorical machinery provides the tools; the correspondence provides the vocabulary.

**Availability.** The reference implementation is open source at <https://github.com/coredipper/operon> (operon-ai on PyPI, MIT license). The five compiler functors, atomic skills catalog, and certificate framework are all included.

## References

- [1] Zhiyu Chen et al. SkillTester: Benchmarking utility and security of agent skills. *arXiv preprint arXiv:2603.28815*, 2026. Comparative quality-assurance harness for agent skills.
- [2] Pablo de los Riscos, Fernando Corbacho, and Michael A. Arbib. Working paper: Towards a category-theoretic comparative framework for artificial general intelligence. *arXiv preprint arXiv:2603.28906*, 2026. Category-theoretic framework (ArchAgents) for comparing agent architectures.
- [3] Qin Liu.  $\lambda_a$ : A typed lambda calculus for LLM agent composition. *arXiv preprint arXiv:2604.11767*, 2026. Formal semantics for agent composition; shows 94.1% of GitHub agent configs are structurally incomplete.
- [4] Yingwei Ma, Yue Liu, Xinlong Yang, et al. Scaling coding agents via atomic skills. *arXiv preprint arXiv:2604.05013*, 2026.
- [5] Lee Marom, Skylar Tibbits, Gioele Zardini, and Markus J. Buehler. A category-theoretic framework from biological mechanics to engineered stimulus-response systems. *arXiv preprint arXiv:2604.26367*, 2026. Defines category Dyn of stimulus-response dynamical systems with subcategories Nat/Art, implementation functor F: Nat to Art, specification space Spec with projection pi, compilation functor E: Spec to Comp; instantiated on hygromorphic pinecone to 4D-printed bilayer pipeline.
- [6] Qianyu Meng, Yanan Wang, Liyi Chen, Wei Wu, Yihang Li, Wenyuan Jiang, Qimeng Wang, Chengqiang Lu, Yan Gao, Yi Wu, and Yao Hu. Agent harness for large language model agents: A survey. <https://www.preprints.org/manuscript/202604.0428/v2>, 2026. Preprints DOI

10.20944/preprints202604.0428.v2. Formalizes the agent harness as a six-component tuple  $H=(E,T,C,S,L,V)$ ; 110+ papers and 23 systems surveyed on a Harness Completeness Matrix; identifies compositional verification as one of nine open challenges. Companion repository: <https://github.com/Gloriaameng/Awesome-Agent-Harness>.

- [7] Richard T. Snodgrass. *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann, 2000. Foundational treatment of valid time, transaction time, and bi-temporal data models.
- [8] Erik Willström et al. Natural-language agent harnesses. *arXiv preprint arXiv:2603.25723*, 2026. Harness behavior as portable, editable natural-language artifacts with explicit contracts.
- [9] Chenyu Zhou, Huacan Chai, Wenteng Chen, et al. Externalization in LLM agents: A unified review of memory, skills, protocols and harness engineering. *arXiv preprint arXiv:2604.08224*, 2026.