EVIDENCE-GUIDED HARNESS EVOLUTION FOR ARTICULATED CAD AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Articulated computer-aided design (CAD) agents rely on modeling knowledge written into their harnesses by hand, and harness evolution could acquire this knowledge from experience without updating the base model. Examining what conventional harness evolution learns, we find that most of its early gain comes from environment knowledge shared across tasks; adding this knowledge to the initial harness by hand recovers the gain. The modeling experience accumulated afterwards helps less reliably, for two reasons: it can reach tasks it does not fit (an *applicability gap*), or the agent can follow it and still carry out the operation incorrectly (a *knowing–doing gap*). The two failures look the same in a task score but call for different revisions. We therefore propose evidence-guided harness evolution, which localizes failures before revising experience. The outer loop tests hypotheses about which experience to add or change through geometric and kinematic checks against training references and targeted trials. Evidence traces each use of an experience through four questions—whether it fit the task, was followed, was carried out correctly, and improved the outcome—and the first that fails decides what to revise. Applicability conditions are narrowed or widened, and operations the agent repeatedly gets wrong become programs checked against local contracts. Starting from an environment-informed harness, so that gains measure what evolution adds beyond it, our method raises the test score from 0.626 to 0.743, above conventional evolution (0.683) and prior harness optimizers. The frozen harness also improves unseen categories, another dataset, and other base models. These results support our view that the scope and realization of experience should be revised with evidence from its use.

1 INTRODUCTION

Computer-aided design (CAD) represents editable 3D objects through structured construction operations, and articulated CAD further specifies how parts are assembled and move relative to one another. Producing such designs requires substantial modeling knowledge, much of which is currently written into an LLM agent's harness by hand. Harness evolution offers a way to acquire this knowledge automatically, by accumulating reusable experience across tasks while the base model stays fixed.

Recent methods that evolve agent prompts and harnesses share a common loop: run the agent, reflect on its trajectories and feedback, edit the harness, and keep edits that improve a validation score (Lin et al., 2026; Lee et al., 2026; Agrawal et al., 2026). Articulated CAD suits this loop: the agent writes CAD programs, and their results can be executed and inspected. It also provides unusually rich feedback. A design decomposes into parts, connections, and motions, so a single execution can succeed on some of them and fail on others. These local successes and failures are themselves a source of reusable experience.

Applying this loop to articulated CAD, we find that much of its early gain comes from environment knowledge, such as interface conventions and stable execution procedures. Adding this knowledge to the initial harness by hand recovers most of the early gain (Section 3). What evolution must provide beyond this point is modeling experience that keeps accumulating across tasks. In our run, however, the modeling experience accumulated afterwards brought only small gains (about 0.05 over the remaining iterations), which raises the question of why it helps later tasks so little.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Our case studies point to one reason: accumulated experience can fail in different ways that look the same in the outcome. One is an *applicability gap*. Environment knowledge is shared across tasks in the same environment, but modeling experience depends on the object and mechanism at hand. Without an explicit scope, it can be provided to tasks whose premises it does not match. Another is a *knowing–doing gap*: the agent follows an applicable experience yet carries out the intended operation incorrectly. We argue that improving reusable experience requires first localizing where in its use a failure arises, and only then deciding what to revise.

We introduce evidence-guided harness evolution, which uses localized evidence to select what to revise. For each hypothesis about what to add or change, the outer loop gathers evidence by checking generated parts and motions against training references and by running targeted trials. Evidence traces each use of an experience through four questions: whether it fit the task, was followed, was carried out correctly, and improved the outcome. The first question that fails selects the revision. A condition that reaches tasks it does not fit is narrowed, and one that misses tasks it fits is widened. An operation that the agent repeatedly carries out incorrectly becomes an executable implementation, checked against a *local contract* that states what any correct implementation must achieve.

Our contributions are:

- A diagnosis of harness evolution for articulated CAD that separates environment knowledge from modeling experience and identifies the applicability and knowing–doing gaps.
- Evidence-guided harness evolution, which localizes failures along the use of an experience, then narrows or widens its applicability condition or turns an operation the agent repeatedly gets wrong into a program checked against a local contract.
- An evaluation from a common environment-informed starting point, where our method raises the test score from 0.626 to 0.743, above conventional evolution (0.683) and prior harness optimizers, and its frozen harness transfers to unseen categories, another dataset, and other base models.

2 Related Work

Harness and Agent Evolution. Automated optimization of LLM agents has broadened from prompts to workflows, agent designs, and full harnesses (Yang et al., 2024a; Agrawal et al., 2026; Zhang et al., 2025b; Hu et al., 2025). Meta-Harness searches over harness code with access to prior candidates, scores, and traces, and Agentic Harness Engineering (AHE) evolves harness components through an observability-driven loop (Lee et al., 2026; Lin et al., 2026). We share this loop; what we add is a diagnosis of individual uses of an experience, which decides whether to revise its content, scope, or realization. Appendix C discusses this line of work in more detail.

Experience Accumulation and Reuse. Experience can be retained within or across tasks. Reflexion keeps verbal reflections between attempts (Shinn et al., 2023). Voyager and CRAFT build libraries of skills or tools, LILO compresses programs into libraries, and LATM lets one model create tools for another (Wang et al., 2024a; Yuan et al., 2024; Grand et al., 2024; Cai et al., 2024). ExpeL, Agent Workflow Memory, Dynamic Cheatsheet, and ACE maintain textual insights, workflows, or playbooks (Zhao et al., 2024; Wang et al., 2024b; Suzgun et al., 2025; Zhang et al., 2025c). Each stores experience in a form fixed in advance, such as text, workflows, or code, and provides it by similarity retrieval or keeps it in context, rather than revising its form or scope from evidence of its later uses.

CAD Generation. CAD generation has moved from learned models of construction sequences (Wu et al., 2021; Khan et al., 2024) to LLMs that write executable CAD programs (Rukhovich et al., 2024; Wang et al., 2025). CADCodeVerify refines such programs with execution and visual feedback, and ToolCAD lets a tool-using agent interact with a CAD engine (Alrashedy et al., 2025; Gong et al., 2026). For articulated objects, ArtiCAD, LAM, and Articraft generate multi-part assemblies in code with specialized agents, checkers, or a domain-specific harness (Shui et al., 2026; Gao et al., 2026; Zhou et al., 2026). We study how such agents can accumulate and revise modeling experience across tasks, reducing the knowledge engineered by hand.

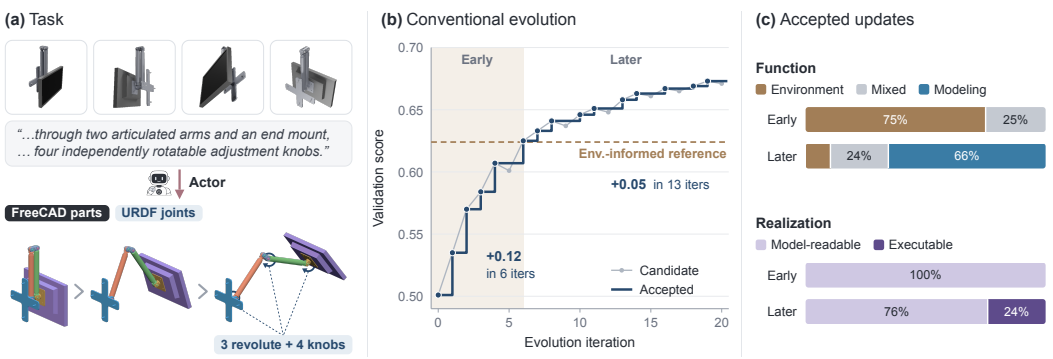


Figure 1: (a) The task: from four views and a description, the Actor writes FreeCAD code with explicit joints. (b) Conventional evolution: validation score of candidate and accepted harnesses; dashed: environment-informed harness. (c) Its accepted updates, early (iterations 1–6) and later.

3 Diagnosing Harness Evolution in Articulated CAD

3.1 Task and Diagnostic Setup

Each task provides four reference images of the target object, taken from different viewpoints and articulation states, and a textual description of its geometry, function, and motion. The agent, which we call the Actor, builds the object as editable parametric geometry in FreeCAD with a URDF-style articulation. Harness evolution runs the Actor on training tasks in batches and revises the harness while the base model stays fixed.

To analyze what the harness learns, we annotate accepted updates according to their function. *Environment knowledge* refers to operational knowledge that is broadly stable across objects, including environment conventions, tool behavior, feedback semantics, and interface-level constraints. *Modeling experience* concerns object- or mechanism-dependent knowledge about how a design should be constructed. Updates that combine both are labeled *mixed*.

3.2 Empirical Diagnosis of Conventional Harness Evolution

We first run conventional harness evolution and track the accepted harness on the validation set. The score rises from 0.501 to 0.625 within six iterations (Figure 1(b)), but reaching its best score of 0.673 takes much longer, with smaller accepted gains and fluctuating candidate scores. Auditing the atomic updates in accepted candidates, we find that the early ones all encode environment or mixed knowledge, mostly environment (Figure 1(c)). As a diagnostic control, we add task-general environment knowledge to the initial harness by hand, leaving modeling strategies unchanged. This environment-informed harness reaches 0.624 without evolution, the level that conventional evolution first reaches at iteration 6. We use this crossing to split the analysis: iterations 1–6 are *early*, and the rest are *later*.

The later phase shifts toward modeling experience, which makes up about two thirds of the later updates, but changes little in how experience is realized: about three quarters of the later updates remain model-readable text in prompts, memories, or skills rather than executable tools or workflow logic. Since real failures are entangled, we examine later trajectories through case studies. In one, a frame-construction convention kept as global guidance led the Actor to add a sill to an open U-shaped frame. In another, the Actor followed the learned hinge-construction guidance, yet the joint moved in the wrong direction.

3.3 Two Gaps in Reusing Experience

Environment knowledge is shared across tasks, so it tends to help wherever it is provided. Modeling experience depends on the object and mechanism, and whether it helps a later task depends on two properties that evolution must maintain for each experience: its *applicability* and its *realization*.

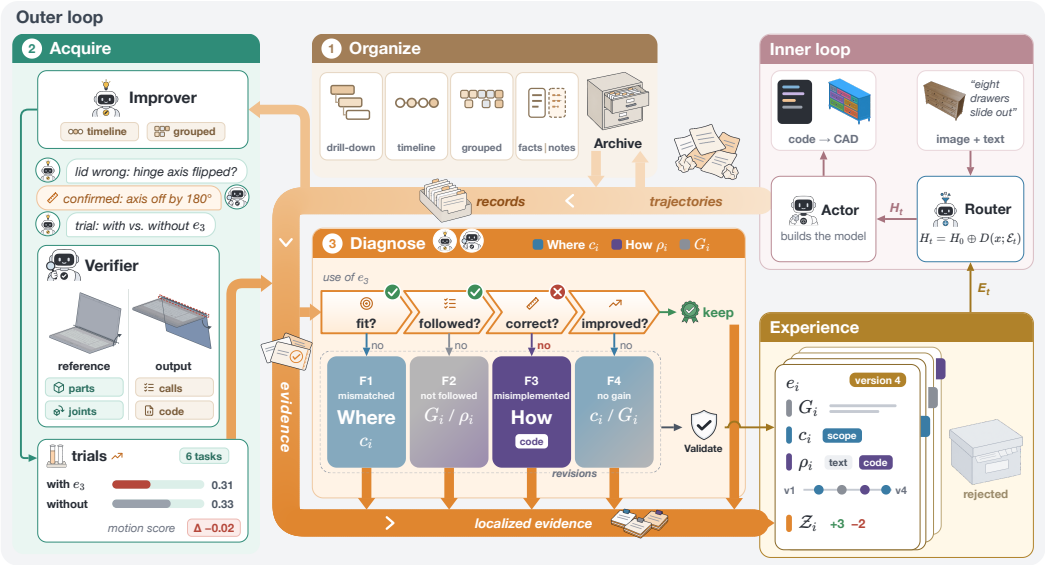


Figure 2: Evidence-guided harness evolution. The inner loop selects experiences by their conditions and solves the task. The outer loop (1) organizes trajectories into records, (2) acquires evidence for hypotheses, and (3) diagnoses each use of an experience; it then adds or revises experiences, changing where they are provided, how they are realized, or what they contain, and stores the evidence with them.

Where: the applicability gap. An experience must reach tasks it fits. It may contain informal cues about when it applies, but evolution does not maintain them as an explicit property. The cues may be absent, too broad, or outdated, and whether the experience applies is left to the Actor's reading of its text. We call the resulting mismatch the *applicability gap*.

How: the knowing–doing gap. An applicable experience that the Actor follows must still be translated into concrete geometric or kinematic operations. This translation can go wrong: the Actor attempts the prescribed operation, but the resulting geometry or motion violates the local requirement. We call this the *knowing–doing gap*.

Both gaps appear in the outcome only as a failed task, yet each needs a different fix; telling them apart requires evidence from how the experience was used.

4 Evidence-Guided Harness Evolution

We call each unit that harness evolution adds to or changes in the harness an experience, whether it is a piece of guidance, a skill, a tool, or a workflow. An inner loop solves tasks with these experiences. An outer loop adds and revises them based on evidence from their uses (Figure 2).

4.1 Harness as Structured Experience

Let M be a fixed base model and $x \sim \mathcal{X}$ an articulated CAD task. A harness H provides the context, reusable knowledge, tools, and execution logic available to M. Running the agent produces a trajectory τ and an executable CAD artifact y,

$$(\tau, y) = \mathrm{Run}(M, H, x), \tag{1}$$

and a fixed evaluator assigns the task-level score $S(x, y)$. Harness evolution changes H while keeping both M and the evaluation protocol fixed.

We represent the persistent state of harness evolution as an evolving set of reusable experience units,

$$\mathcal{E}_t = \{e_i\}_{i=1}^{N_t}, \tag{2}$$

where each experience is

$$e_i = (G_i, c_i, \rho_i, \mathcal{Z}_i). \tag{3}$$

Here, G_i is the reusable content captured by the experience, c_i describes its applicability, ρ_i specifies how it is realized during execution, and $\mathcal{Z}_i$ stores evidence accumulated from its uses and from targeted trials. The realization ρ_i may be model-readable, executable, or hybrid, instantiated through prompts, memories, skills, tools, or workflows. An experience is thus a unit of the harness itself: a model-readable experience changes what the Actor reads, and an executable one changes what the harness does.

For each task, this experience state is instantiated into a task-specific harness:

$$H_t(x) = H_0 \oplus D(x; \mathcal{E}_t), \tag{4}$$

where H_0 is the initial harness, D selects applicable experiences and deploys their current realizations, and $\oplus$ adds them to H_0; the evidence $\mathcal{Z}_i$ stays in the outer loop.

4.2 Evidence-Guided Evolution Loop

Evolution alternates between an inner loop that solves tasks and an outer loop that adds and revises experience from evidence (Figure 2). In the inner loop, a Router judges each condition c_i against the full task materials, the selected experiences are assembled with H_0 into $H_t(x)$ (Eq. 4), and the Actor solves the task within $H_t(x)$.

Conventional harness evolution reflects on recent trajectories, edits the harness, and keeps edits that improve validation performance (Lin et al., 2026; Lee et al., 2026; Agrawal et al., 2026). We instead revise experience only on evidence, following the chain:

$$(\text{history}, \mathcal{E}_t) \rightarrow \text{hypothesis} \rightarrow \text{investigation}^{\circlearrowleft} \rightarrow \text{localized evidence} \rightarrow \text{update} \rightarrow \text{validation}.$$

A new experience passes through the same chain as a revision. After each batch of training tasks, the Improver reads the history and the current experiences and proposes hypotheses about which experience to add or change and why a use failed. The Verifier, a second agent, investigates each hypothesis against executions and artifacts and reports supporting or opposing evidence. The resulting evidence is localized to specific uses of experiences, and on this basis the Improver adds, revises, or keeps experience. A candidate harness is accepted only if it improves validation performance. The evidence itself is built in three steps, shown as (1)–(3) in Figure 2.

Organizing evidence. Evolution produces far more trajectories and artifacts than an agent can read, so a deterministically built archive turns them into records that can be cited as evidence. The agents query it through the file system and dedicated tools: views drill down from batch overviews to individual tool calls and raw records, follow a task along its timeline, or group records across tasks by tool, error type, or experience. Aggregate statistics report how many records they rest on, so that the agents do not over-read small samples. The agents' interpretations are kept apart from these facts and cite the records they rest on, and counterexamples and failed attempts are kept alongside supporting evidence.

Acquiring evidence. A trajectory records what happened; it becomes evidence only when examined against a hypothesis in a way that could support or refute it. Evidence comes from two sources. First, the Verifier checks artifacts against the training references, which are part-segmented meshes and articulation specifications. It locates the part or joint in question, measures it, and compares the measurement with the reference or with a stated local requirement (Section 4.4). Second, the agents also compare executions across tasks. They compare trajectories for errors that recur across tasks, and run targeted trials: a task with an experience, without it, or with a modified version.

Evidence-guided diagnosis. A low score can originate at different points in the use of an experience, and each point calls for a different revision. The Improver and Verifier therefore trace each use through four questions: whether the experience fits the task, whether the Actor followed it, whether it was carried out correctly, and whether it improved the outcome. Each is answered from acquired evidence rather than from the trajectory alone. The first question answered negatively labels the use:

- **F1, mismatched:** the Router provided the experience, but its premises do not hold for the task, judged from the task materials. The experience may be sound, but its condition reaches a task it should not; this is the applicability gap, and the revision targets c_i.
- **F2, not followed:** the experience fits, but the Actor did not act on it. The guidance did not reach the Actor's actions; possible causes include unclear guidance, missing inputs, or a carrier that is hard to act on, and the revision rewrites G_i or moves it to an easier carrier.
- **F3, misimplemented:** the Actor followed the experience but carried out the operation incorrectly. This is the knowing–doing gap, and the revision targets the realization ρ_i.
- **F4, no gain:** the experience fits, was followed, and was carried out correctly, yet the task did not improve. If such uses concentrate in one kind of task, the condition is broader than where the experience helps and is narrowed; otherwise its content G_i is reconsidered.

A use that passes all four questions is positive evidence for the experience. Labeled uses form the evidence $\mathcal{Z}_i = \mathcal{Z}_i^+ \cup \mathcal{Z}_i^-$ of each experience, where $\mathcal{Z}_i^-$ holds the uses that failed one of the questions. Each use links to the records it rests on. A use is labeled at the first question that evidence answers negatively; if an earlier question cannot be answered, the use is kept without a label. If mismatches or uses without gain concentrate in one kind of task, they guide a revision of where the experience is provided (Section 4.3); if a knowing–doing gap recurs on one operation, it guides a revision of how the operation is realized (Section 4.4).

4.3 Revising Applicability: Where

In textual experience, the cues about when to use it sit inside the content and are not tracked as the experience evolves. We therefore keep the condition c_i as an explicit, versioned field, separate from G_i, that describes the tasks to which e_i is worth providing. The condition is a hypothesis about scope. The Improver initializes it from the source task and its evidence, the Router interprets it against the full task materials, and later evidence revises it.

Depending on what the diagnosed uses show, the Improver can narrow a condition or widen it, and the Verifier checks each proposed change against those uses. Mismatches (F1), or uses without gain (F4) that concentrate in one kind of task, suggest that a condition is too broad; the Improver then narrows it, adds an exception, or splits it where its subcases conflict. Positive evidence outside the condition suggests that it is too narrow. Such evidence comes from a trial of an unselected experience, or from the Actor succeeding with the same strategy without receiving the experience; in the second case, the Improver first runs a trial with the experience provided and widens the condition only if the trial helps. Since c_i is separate from G_i and ρ_i, these revisions change which tasks receive an experience without changing its content or realization.

4.4 Revising Realization: How

Realization determines how the content G_i becomes behavior. When the Actor does not act on an applicable experience (F2), the Improver keeps the fix model-readable: it rewrites the guidance or moves it to another carrier, a skill or a workflow, that is easier to act on. When the Actor follows the experience but carries out the operation incorrectly (F3) and this knowing–doing gap recurs on the same operation, the Improver tests whether an executable realization is more reliable than the text.

Executable realization. Not every piece of guidance can become a program. The Improver converts an operation only when three conditions hold. First, its interface is explicit, so it can be written as a function $f_i : \mathcal{I}_i \to \mathcal{O}_i$ from inputs to outputs. Second, it is general: one procedure serves a whole set of valid inputs rather than a single case. Third, its result is checkable: a test $V_i : \mathcal{I}_i \times \mathcal{O}_i \to \{0, 1\}$ can decide from an input and an output whether the operation was done correctly. The third condition matters most. Once the Actor calls a program, a failed task no longer shows whether the program or the Actor is wrong; only a check on the operation itself can. Whether and when to use the operation, and with which inputs, remain decisions of the model; the program only carries the operation out.

Local contracts. The check V_i comes from a local contract $K_i = (P_i, R_i)$, which the Improver writes and the Verifier reviews in the spirit of design by contract. The precondition P_i states which

inputs the operation is meant to handle. The requirement R_i states what a correct output must satisfy for such an input. An implementation is correct if every input that satisfies P_i yields an output that satisfies R_i:

$$P_i(u) \;\Rightarrow\; R_i\big(u, f_i(u)\big).$$

The contract states what the operation must achieve, not how, so the Actor's own implementation and the program are judged by the same standard.

The same contract is used three times. First, it confirms the gap: if the input satisfies P_i but the Actor's output violates R_i, the Actor carried out the operation incorrectly. Second, it tests the program before adoption. Rerunning whole tasks would add noise from unrelated steps, so the Improver replays only the failed step. It restores the context of that step from a training run and lets the Actor continue twice, once with the program and once with the textual guidance. The contract checks the program's output on the replayed step, and comparing the two branches shows whether the Actor uses the program well and whether it helps. The program is adopted only if it also improves validation performance. Third, after deployment it assigns responsibility: a violated result under satisfied preconditions points to the program, and a violated precondition to how the model chose or used the operation, sending the evidence back to the guidance or the condition; if the contract holds but the task fails, the agents examine the operation's selection, its interaction with later steps, and failures outside the contract. Later evidence can also narrow the program's scope or hand the operation back to the model. The goal is to place each part of an experience where it is executed most reliably.

5 Experiments

5.1 Experimental Setup

Tasks and protocol. We construct 500 tasks from Artiverse objects (Iliash et al., 2026) and split them into 200 training, 100 validation, and 200 test tasks. The objects contain 6,609 parts and 6,109 joints in total. We also test transfer to PartNet-Mobility objects (Xiang et al., 2020). Training tasks are processed in batches of 10, with one outer-loop round after each batch, giving 20 rounds. The Actor uses GPT-5.6 Luna, and the outer-loop agents use GPT-5.6 Sol. All evolving methods start from the environment-informed harness and share the same models, batches, rounds, and acceptance rule (Appendix E).

Metrics. One evaluation program checks each generated model against requirements annotated from the reference object, in three aspects scored in $[0, 1]$: *Struct* (parts, dimensions, and fixed connections), *Motion* (joints and the motion they produce), and *Assembly* (layout, contact, and interference across poses). The *program score* (Score in the tables), used for validation and acceptance, is their mean, and a failed build scores 0. We also report the surface *F-score* at a threshold of 1% of the reference bounding-box diagonal and a 0–5 rating by an agent judge (Zhuge et al., 2024) based on GPT-6 Astra on the test set only (Appendices E and F).

Baselines. Two fixed harnesses serve as references: the minimal harness, the starting point of Section 3, and the environment-informed harness, which adds the environment knowledge by hand and is the starting point of all evolving methods. The evolving baselines are GEPA, which evolves prompts (Agrawal et al., 2026); Meta-Harness, which searches over harness code (Lee et al., 2026); and conventional evolution, the loop of Section 3, designed for this task but without evidence guidance. LAM is a multi-agent pipeline designed by hand for articulated object generation (Gao et al., 2026).

5.2 Main Results

Table 1 reports the test results. Our method has the highest mean on every metric in Table 1. Its gain is largest in Motion and Assembly, whose errors arise at individual joints and part relations. These are the knowing–doing gaps our method targets, and rolling back executable realizations lowers Motion the most (Table 4).

Table 1: Results on the 200 Artiverse test tasks. Subscripts: change from the environment-informed harness, computed before rounding; bold: best; underlined: second best.

Method	Score	Judge	Struct	Motion	Assembly	F-score
Minimal harness	0.524	3.66	0.784	0.379	0.409	0.450
Env.-informed harness	0.626	3.95	0.859	0.510	0.509	0.486
GEPA	$0.632_{+.006}$	$4.08_{+.13}$	$0.853_{-.006}$	$0.505_{-.005}$	$0.537_{+.029}$	$0.488_{+.002}$
Meta-Harness	$0.674_{+.048}$	$\underline{4.31}_{+.36}$	$\underline{0.877}_{+.017}$	$0.567_{+.057}$	$0.578_{+.070}$	$\underline{0.504}_{+.018}$
Conventional evolution	$\underline{0.683}_{+.057}$	$4.27_{+.32}$	$0.856_{-.003}$	$\underline{0.588}_{+.078}$	$\underline{0.605}_{+.096}$	$0.501_{+.015}$
LAM (hand-designed)	0.666	4.17	0.858	0.562	0.578	0.495
Evidence-guided (ours)	$\mathbf{0.743}_{+.117}$	$\mathbf{4.52}_{+.57}$	$\mathbf{0.903}_{+.044}$	$\mathbf{0.662}_{+.152}$	$\mathbf{0.664}_{+.155}$	$\mathbf{0.524}_{+.038}$

Table 2: Program score by test tier (134, 33, and 33 tasks) and on PartNet-Mobility (100 tasks). Subscripts: change from the environment-informed harness.

Method	Artiverse test			PartNet-Mobility
	Seen cat.	Unseen cat.	Unseen super-cat.	
Env.-informed (fixed)	0.641	0.634	0.558	0.598
GEPA	$0.636_{-.004}$	$0.704_{+.069}$	$0.543_{-.015}$	$0.582_{-.016}$
Meta-Harness	$0.672_{+.031}$	$0.772_{+.138}$	$0.586_{+.028}$	$0.611_{+.013}$
Conventional	$0.694_{+.054}$	$0.719_{+.084}$	$0.601_{+.044}$	$0.622_{+.024}$
Ours	$\mathbf{0.748}_{+.107}$	$\mathbf{0.779}_{+.145}$	$\mathbf{0.687}_{+.129}$	$\mathbf{0.641}_{+.043}$

Gains beyond environment knowledge. How much a harness improves depends on where it starts. From a weak start, much of the gain is environment knowledge (Section 3), which can be acquired by hand or in the first rounds of evolution. We therefore start all evolving methods from the environment-informed harness, so that their gains measure what evolution adds beyond it. From there, GEPA gains 0.006, and Meta-Harness and conventional evolution gain 0.048 and 0.057, about half of our 0.117, although both end well above the minimal harness. Evaluated from a weak start, this difference would be hidden behind environment adaptation. This suggests that our method is better at accumulating modeling experience that carries over to later tasks, which is what evolution must provide once environment knowledge is in place.

Evolution versus hand design. LAM, designed by hand for articulated object generation, scores below the three methods that evolve harness code. Its components were built for a setting whose inputs, environment, and data differ from ours, which may limit how well they transfer; harness evolution instead acquires such knowledge in each new setting.

5.3 Generalization of the Evolved Harness

We freeze each evolved harness and split the test tasks into three tiers by their distance from the training and validation data: seen categories (134 tasks), unseen categories within seen super-categories (33), and unseen super-categories (33). Categories differ in difficulty, so the tiers start from different scores; we therefore compare gains over the environment-informed harness rather than scores across tiers (Table 2). Our harness gains the most on every tier, including unseen super-categories, where conventional evolution gains less than half as much and GEPA falls below its starting point. On PartNet-Mobility, every gain shrinks, but ours remains the largest. These gains support transfer beyond the categories and the dataset used for evolution. This suggests that much of the accumulated experience is reusable modeling knowledge, not only fixes for the tasks seen during evolution.

Deployed on three other base models without further evolution (Table 3), the harness evolved with GPT-5.6 Luna improves each by 0.066 to 0.091 although they start from 0.375 to 0.584, so the evolved experience can be reused by models of different ability.

Table 3: Transfer to other base models without further evolution.

Base model	Env.	Ours
Gemini 3.7 Flash	0.584	$0.657_{+.073}$
DeepSeek V4 Flash	0.375	$0.466_{+.091}$
Claude Sonnet 5	0.447	$0.513_{+.066}$

Table 4: Ablations on the test tasks.

Variant	Struct	Motion	Assembly	Score
Full	0.903	0.662	0.664	0.743
Conventional evolution	0.856	0.588	0.605	0.683
Re-evolved w/o conditions	0.864	0.608	0.620	0.697
Rolled back to text	0.836	0.543	0.580	0.653

5.4 Ablations

Table 4 compares our method with three variants. Conventional evolution uses none of the evidence-guided mechanisms. The second variant reruns evolution without conditions or the Router, so that every task can draw on all experiences. The third restores each executable realization in our final state to its last model-readable version.

Evidence-guided outer loop. Conventional evolution receives the same materials and base models, and its Improver may inspect references or write tools, but nothing asks it to form hypotheses, localize failures, or maintain conditions and realizations. It ends 0.060 below our method. With access held equal, the gap reflects the policy that directs how the outer loop uses these materials, here the evidence-guided design as a whole.

Explicit scope. Evolving without conditions and the Router lowers the score by 0.046. A likely reason is that the Actor must then judge for itself which experience applies, and experience whose premises do not hold can reach the task. This grows harder as experiences accumulate and compete for the Actor's context, where unrelated or conflicting guidance may interfere.

Executable realizations. Rolling executable realizations back to their earlier textual versions lowers the score from 0.743 to 0.653, below conventional evolution, which keeps its own executable updates. Many CAD operations are precise geometric computations, such as placing a joint axis: text can state what to compute, but a program computes it the same way every time and can be checked against its contract. With the small gain of GEPA, this suggests that in articulated CAD, how an experience is realized matters as much as what it says.

5.5 Discussion

The loop needs signals that locate failures. In exploratory runs whose only signals were Chamfer distance, F-score, and a simple collision test, evolution made little progress: such signals report that a model is wrong but not where or why. The Verifier's comparisons with training references give signals at the level of parts, joints, and operations, and our gains concentrate where these signals are most specific, in Motion and Assembly. Harness evolution for software may benefit for the same reason, since unit tests localize failures there; elsewhere, such checks may need to be built first.

What the harness does not change. Evolution makes several recurring operations more reliable, but complex curved surfaces remain difficult: they are still often approximated by boxes or cylinders, even when the harness explains how FreeCAD builds such shapes, possibly a limit of the base model. Raising this ability may require better modeling interfaces for agents, much as the agent–computer interface matters in software engineering (Yang et al., 2024b); harness evolution has not yet produced such designed changes, which we leave to future work.

6 Conclusion

In our study of harness evolution for articulated CAD, much of the early gain came from environment knowledge, and later modeling experience failed in ways that look alike in the outcome. Evidence-guided harness evolution localizes these failures and revises the content, applicability, or realization of an experience. It outperforms conventional evolution and prior harness optimizers and transfers to unseen categories, another dataset, and other base models. It may carry over to other domains whose artifacts can be checked part by part; limitations are discussed in Appendix B.

AI USE STATEMENT

We used generative AI to draft and polish the manuscript, search for related literature, and assist in preparing some figures. We also used GPT-5.5 to write task descriptions and annotate evaluation requirements from existing dataset records, as described in Appendix E. Human reviewers checked the task descriptions. We manually annotated the requirements of 30 tasks and reviewed the annotation protocol and evaluation program before using GPT-5.5 to annotate the remaining tasks. The use of language models within our method and experiments is described in Section 5.1 and Appendix E. The authors take responsibility for the final manuscript, results, and accompanying artifacts.

ETHICS STATEMENT

Our experiments use objects from the publicly available Artiverse and PartNet-Mobility datasets. Internal team members performed the human evaluation of generated models; we collected no personal or sensitive data. Our evaluation concerns geometry, articulation, and assembly, and does not establish the safety of generated designs for physical fabrication or deployment. Such applications require independent engineering review and validation.

REPRODUCIBILITY STATEMENT

Section 4 and Appendix D.1 describe the method and outer-loop implementation. Section 5.1 and Appendix E document the datasets, splits, annotation procedure, evaluation metrics, baselines, models, and execution budgets. Appendix F discusses the evaluation protocol, and Appendix G.2 reports additional analyses of the evolution process. We will release the source code upon acceptance.

REFERENCES

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective Prompt Evolution Can Outperform Reinforcement Learning. In *International Conference on Learning Representations (ICLR)*, 2026.

Kamel Alrashedy, Pradyumna Tambwekar, Zulfiqar Zaidi, Megan Langwasser, Wei Xu, and Matthew Gombolay. Generating CAD Code with Vision-Language Models for 3D Designs. In *International Conference on Learning Representations (ICLR)*, 2025.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large Language Models as Tool Makers. In *International Conference on Learning Representations (ICLR)*, 2024.

Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-Referential Self-Improvement Via Prompt Evolution. *arXiv preprint arXiv:2309.16797*, 2023.

Yipeng Gao, Yunhao Ge, Peilin Cai, Daniel Seita, and Laurent Itti. LAM: Language Articulated Object Modelers. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2026.

Yifei Gong, Xing Wu, Wenda Liu, and Kang Tu. TOOLCAD: Exploring Tool-Using Large Language Models in Text-to-CAD Generation with Reinforcement Learning. *arXiv preprint arXiv:2604.07960*, 2026.

Gabriel Grand, Lionel Wong, Maddy Bowers, Theo X. Olausson, Muxin Liu, Joshua B. Tenenbaum, and Jacob Andreas. LILO: Learning Interpretable Libraries by Compressing and Documenting Code. In *International Conference on Learning Representations (ICLR)*, 2024.

Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. EvoPrompt: Connecting LLMs with Evolutionary Algorithms Yields Powerful Prompt Optimizers. In *International Conference on Learning Representations (ICLR)*, 2024.

Shengran Hu, Cong Lu, and Jeff Clune. Automated Design of Agentic Systems. In *International Conference on Learning Representations (ICLR)*, 2025.

Denys Iliash, Jiayi Liu, Egor Fokin, Qirui Wu, Ali Mahdavi-Amiri, Manolis Savva, and Angel X. Chang. Artiverse: A Diverse and Physically Grounded Dataset for Articulated Objects. *arXiv preprint arXiv:2605.24403*, 2026.

Mohammad Sadil Khan, Sankalp Sinha, Talha Uddin Sheikh, Didier Stricker, Sk Aziz Ali, and Muhammad Zeshan Afzal. Text2CAD: Generating Sequential CAD Models from Beginner-to-Expert Level Text Prompts. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines. In *International Conference on Learning Representations (ICLR)*, 2024.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-Harness: End-to-End Optimization of Model Harnesses. *arXiv preprint arXiv:2603.28052*, 2026.

Jiahang Lin, Shichun Liu, Chengjun Pan, Lizhi Lin, Shihan Dou, Zhiheng Xi, Xuanjing Huang, Hang Yan, Zhenhua Han, Tao Gui, and Yu-Gang Jiang. Agentic Harness Engineering: Observability-Driven Automatic Evolution of Coding-Agent Harnesses. *arXiv preprint arXiv:2604.25850*, 2026.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-Refine: Iterative Refinement with Self-Feedback. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

Danila Rukhovich, Elona Dupont, Dimitrios Mallis, Kseniya Cherenkova, Anis Kacem, and Djamila Aouada. CAD-Recode: Reverse Engineering CAD Code from Point Clouds. *arXiv preprint arXiv:2412.14042*, 2024.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (eds.), *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL `http://papers.nips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html`.

Yuan Shui, Yandong Guan, Zhanwei Zhang, Juncheng Hu, Jing Zhang, Dong Xu, and Qian Yu. ArtiCAD: Articulated CAD Assembly Design via Multi-Agent Code Generation. *arXiv preprint arXiv:2604.10992*, 2026.

Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic Cheatsheet: Test-Time Learning with Adaptive Memory. *arXiv preprint arXiv:2504.07952*, 2025.

Maxim Tatarchenko, Stephan R. Richter, René Ranftl, Zhuwen Li, Vladlen Koltun, and Thomas Brox. What Do Single-view 3D Reconstruction Networks Learn? In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An Open-Ended Embodied Agent with Large Language Models. *Transactions on Machine Learning Research*, 2024a.

Ruiyu Wang, Shizhao Sun, Weijian Ma, and Jiang Bian. CAD-Tokenizer: Towards Text-based CAD Prototyping via Modality-Specific Tokenization. *arXiv preprint arXiv:2509.21150*, 2025.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent Workflow Memory. *arXiv preprint arXiv:2409.07429*, 2024b.

Rundi Wu, Chang Xiao, and Changxi Zheng. DeepCAD: A Deep Generative Network for Computer-Aided Design Models. In *IEEE/CVF International Conference on Computer Vision (ICCV)*, 2021.

Fanbo Xiang, Yuzhe Qin, Kaichun Mo, Yikuan Xia, Hao Zhu, Fangchen Liu, Minghua Liu, Hanxiao Jiang, Yifu Yuan, He Wang, Li Yi, Angel X. Chang, Leonidas J. Guibas, and Hao Su. SAPIEN: A SimulAted Part-based Interactive ENvironment. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large Language Models as Optimizers. In *International Conference on Learning Representations (ICLR)*, 2024a.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024b.

Lifan Yuan, Yangyi Chen, Xingyao Wang, Yi R. Fung, Hao Peng, and Heng Ji. CRAFT: Customizing LLMs by Creating and Retrieving from Specialized Toolsets. In *International Conference on Learning Representations (ICLR)*, 2024.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. TextGrad: Automatic "Differentiation" via Text. *arXiv preprint arXiv:2406.07496*, 2024.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin Gödel Machine: Open-Ended Evolution of Self-Improving Agents. *arXiv preprint arXiv:2505.22954*, 2025a.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating Agentic Workflow Generation. In *International Conference on Learning Representations (ICLR)*, 2025b.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic Context Engineering: Evolving Contexts for Self-Improving Language Models. *arXiv preprint arXiv:2510.04618*, 2025c.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM Agents Are Experiential Learners. In *AAAI Conference on Artificial Intelligence*, 2024.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2023.

Matt Zhou, Ruining Li, Xiaoyang Lyu, Zhaomou Song, Zhening Huang, Chuanxia Zheng, Christian Rupprecht, Andrea Vedaldi, and Shangzhe Wu. Articraft: An Agentic System for Scalable Articulated 3D Asset Generation. *arXiv preprint arXiv:2605.15187*, 2026.

Mingchen Zhuge, Changsheng Zhao, Dylan Ashley, Wenyi Wang, Dmitrii Khizbullin, Yunyang Xiong, Zechun Liu, Ernie Chang, Raghuraman Krishnamoorthi, Yuandong Tian, Yangyang Shi, Vikas Chandra, and Jürgen Schmidhuber. Agent-as-a-Judge: Evaluate Agents with Agents. *arXiv preprint arXiv:2410.10934*, 2024.

A More Discussion and Outlook

Validation as a bottleneck. Harness evolution methods either accept every update and observe later tasks, or, as we do, accept an update only if it improves a validation set. Validation is expensive: each candidate requires a full pass, and interacting updates multiply the passes needed to separate their effects. A gain on the validation set is also not proof of a genuine improvement, since it can reflect evaluation noise, overfitting to the validation set through repeated selection, or a change in the input format that shifts the model's behavior. A larger validation set reduces sampling noise at a higher cost, but not adaptive overfitting. Our Verifier already chooses targeted trials; a natural extension is to let it choose the validation as well, running the tasks an update is likely to affect, tasks it should not affect, and likely counterexamples, instead of a fixed set, as long as acceptance also rests on tasks the agents cannot choose. This would give the agents responsibility for selecting acceptance tests and could make validation both cheaper and more informative.

Evolving the evolution policy. The policy under which the optimizer runs (Section 5.4) is designed by people and stays fixed during evolution, which bounds what the optimizer can do. A system could instead treat its own procedure as an object of revision: notice that some of its investigative steps are wasteful or misleading, and change how it gathers evidence, what it records, and when it revises. Work on self-referential improvement points in this direction (Zhang et al., 2025a; Hu et al., 2025), and harness evolution offers a setting where such changes can be evaluated against task outcomes.

Decomposition and cost. Splitting the task into more stages and roles did not help in our experiments: it raised the cost more than the score. Most parts of a 3D design depend on the whole object, so each stage still needs a global view, and only a few subproblems separate cleanly. A minimal harness can produce the core code and joints of a task in a few tens of thousands of tokens, and the harness that evolution produces adds experience and tools rather than stages. We report this as an observation on our task, not as a judgment on multi-agent designs in general.

Active investigation and self-maintained records. Harness evolution resembles scientific work: observing executions, forming hypotheses, testing them, and summarizing what holds. Our comparison with conventional evolution suggests that an outer loop that investigates actively gathers more useful evidence than one that reflects on the trajectories it is given, and passive reflection becomes more expensive as the history grows. Organizing the history by rules, as our archive and AHE do (Lin et al., 2026), keeps it searchable, but the records and summaries still grow with every round and eventually exceed what an agent can use. Deciding which records to keep, which to abstract, and which to discard changes as evolution proceeds and is difficult to fix in advance. It likely requires the agents to maintain their own records. In our experiments, current models did not yet sustain this kind of self-directed maintenance, so it remains an open problem.

B Limitations

Our study focuses on articulated CAD; whether localizing failures before revising experience brings similar gains in other domains is a direction for future work. The Verifier relies on the reference meshes and articulation specifications of training tasks, so domains without such references or feedback would need other sources of evidence. The public baselines were designed for other settings and are adapted to ours. Routing relies on the Router's semantic reading of conditions, which can itself err.

C Extended Related Work on Harness and Agent Evolution

Before harnesses, the prompt was the main object of optimization. OPRO uses an LLM to optimize its own instructions (Yang et al., 2024a), EvoPrompt and Promptbreeder evolve prompts with evolutionary operators (Guo et al., 2024; Fernando et al., 2023), TextGrad propagates textual feedback through a computation graph (Yuksekgonul et al., 2024), DSPy compiles declarative pipelines of model calls (Khattab et al., 2024), and GEPA combines reflection on rollouts with Pareto-based selection (Agrawal et al., 2026). AFlow and ADAS extend the search to workflows and agent designs

expressed as code (Zhang et al., 2025b; Hu et al., 2025). AlphaEvolve evolves programs with a coding agent, and the Darwin Gödel Machine evolves an agent's own code (Novikov et al., 2025; Zhang et al., 2025a). Work on agent–computer interfaces, such as SWE-agent, shows how strongly the layer around the model shapes its performance (Yang et al., 2024b), which is the layer that harness optimization changes. Within a single task, Self-Refine and Reflexion improve behavior from self-feedback or verbal reflection (Madaan et al., 2023; Shinn et al., 2023). Across these methods, the optimizer learns from scores, reflections on traces, or the execution history, which tell it that a task failed and often what happened; we add a diagnosis of individual uses of an experience.

D Method Details

D.1 Implementation of the Outer Loop

The Improver and Verifier are coding agents that run on the Codex agent runtime and are driven through its Python SDK, in the same way that Meta-Harness uses a coding agent as its proposer (Lee et al., 2026). Each outer-loop round is limited to one hour of wall-clock time; within a round, token usage is not limited, and the agents decide how far to investigate: the investigation may continue, change direction, or end without an update. After evolution, the accepted state is frozen for testing. The two agents share a workspace and exchange findings through files. The Improver's hypotheses concern a new experience, a change to the content, condition, or realization of an existing one, or the effect of an experience on a task; the Verifier checks both the proposed change and its supporting interpretation against evidence. Evidence from rejected candidates is retained as counterexamples or as support for a narrower experience. Trajectories, source code, artifacts, and feedback are archived for the outer loop. Raw outer-loop materials, such as training references, scores, and investigation records, never enter the Actor's context; only accepted experiences are deployed.

D.2 Archive

The archive is built by deterministic rules and indexes only training executions, so test tasks and private evaluation details never enter it. As in prior systems that expose execution history to the optimizer (Lin et al., 2026; Lee et al., 2026), the agents access it through the file system and dedicated query tools. Views drill down from batch overviews to individual tool calls and raw records, follow a task along its timeline, or group records across tasks by tool, error type, or experience. Statistics report their sample sizes and missing data, distinguishing counts of calls, episodes, and distinct tasks, so that repeated references to the same material do not inflate the evidence.

D.3 Reference Checks and Targeted Trials

Training tasks provide part-segmented reference meshes and articulation specifications, including joint types, parent–child structure, axes, origins, and motion ranges. They contain no ground-truth FreeCAD programs or modeling sequences, so the outer loop must still find operations that produce the required geometry and motion. The Actor can check its output only against the input images and text. The Verifier localizes the part, joint, or operation in question, binds the quantities to check, such as a child part, a witness point (a point on the child part whose trajectory is compared), and joint states, matches parts to the reference from renderings or with provided or self-written tools, and compares the measurements with the reference or with a stated local requirement. Evidence thus concerns geometry and motion; the Actor's code is read only to trace which operation produced a part. Besides running a task with or without an experience or with a modified version, trials can test an experience that the Router did not select. Local tool changes are tested by replaying a single step and broader guidance by rerunning the task.

D.4 Diagnosis Criteria

Fit asks whether the premises of the experience hold for the task, judged from the task materials and the reference independently of the outcome. Whether the Actor followed the experience is judged by comparing its actions with the guidance, or directly from call records for tool-based realizations. Correctness is measured through checks bound to the identified parts, joints, and poses, and improvement is established by a controlled comparison. If a part cannot be matched or a check cannot

be run, the use is kept without a label rather than counted as a failure. For F1, the agents examine the condition and the Router's reading of it. For F2, they check whether the guidance is clear, the needed inputs are available, and the realization is easy to act on, and they may also revise the usage instructions. For F3, they separate wrongly chosen inputs from a faulty implementation. For F4 outside an identifiable kind of task, they revisit the content, task-specific inputs, and interactions with other experiences. The labels separate cases that look the same in a score: F1 can actively mislead the Actor, whereas an F2 use often leaves its behavior unchanged. We assess the reliability of these labels in Appendix G.3.

D.5 Conditions and Routing

A condition is a versioned description of the tasks for which an experience is worth providing, kept separate from the content. Widening a condition needs positive evidence outside it, and such evidence is scarce without trials, because an experience that is never selected is never tested outside its condition. Experiences with overlapping conditions may be merged, and an experience whose condition is no longer supported by evidence may be withdrawn from routing. Old versions are kept, so every revision can be traced to the evidence behind it and reverted.

D.6 Local Contracts

The precondition P_i differs from the condition c_i: c_i decides at the start of a task whether an experience is provided, whereas P_i decides during execution whether an input is one the operation handles. Besides P_i and the requirement R_i of Section 4.4, a contract may state properties W_i that the operation must leave unchanged outside its scope, which we check together with R_i. Checkers test every clause of the contract on artifacts by measuring geometry, joint parameters, or motion. Contracts combine a structured shell with semantic text. Both agents may propose a contract from their investigation; the Improver writes it into the experience, and the Verifier reviews how each part maps to a check. The Actor's implementation and a candidate program are checked against the same version of the contract. Contracts, checkers, their bindings to artifacts, and trial inputs are versioned separately, and a change to any of them triggers a review of the pass or fail results that depend on it.

D.7 Agent Roles

The Actor receives the task materials and the harness $H_t(x)$ and writes and executes FreeCAD code. The Router uses the same base model as the Actor, GPT-5.6 Luna; it receives the task materials and the conditions of all experiences and returns the experiences to provide. The Improver receives the archive and the experience state and proposes and writes revisions; the Verifier receives the same archive, the training references, and the Improver's hypotheses, and reports evidence.

E Experimental Details

E.1 Datasets

Artiverse (Iliash et al., 2026) provides part-segmented meshes and articulation specifications but no textual descriptions, so GPT-5.5 writes a description of each object's geometry, function, and motion from the dataset records, and a human checks it; each description is paired with four reference renderings from different viewpoints and articulation states. The split into 200 training, 100 validation, and 200 test tasks is fixed before evolution (Table 5). Test tasks fall into three tiers by their distance from the training and validation data: seen categories (134 tasks), unseen categories within seen super-categories (33), and unseen super-categories (33). For transfer, we select 100 PartNet-Mobility objects (Xiang et al., 2020), choosing categories that do not appear in our Artiverse splits, and prepare them with the same pipeline.

E.2 Requirement Annotation and Program Metrics

We manually annotated the checkable requirements of 30 tasks and, from them, built a single evaluation program with a fixed requirement format. After the annotations and the program were reviewed and found consistent, GPT-5.5 annotated the remaining tasks with the same standard from the dataset

Table 5: Dataset splits.

Split	Tasks	Parts	Joints
Training	200	2,577	2,377
Validation	100	1,470	1,370
Test	200	2,562	2,362
Total	500	6,609	6,109

records. This step extracts fixed information, such as parts, dimensions, joints, and motion ranges, from the reference data; the annotations are generated once and frozen before evolution. The evaluator then measures each generated model and its URDF articulation against these requirements and the reference geometry, without LLM or human scoring. *Struct* aggregates required parts, dimensions recoverable from the reference data, fixed connections, and major cavities, weighted by importance. *Motion* matches movable units, takes for each unit the limiting quality among its joint checks (type, axis, range, reachability of the public poses, and the resulting motion), and aggregates by group. *Assembly* combines the visible layout with losses from collisions and broken connections across poses. Numerical deviations within a fixed tolerance receive full credit and are penalized continuously beyond it; discrete errors such as a missing part or a wrong joint type zero the corresponding unit. Aggregating by group with preset importance keeps many correct small parts from masking an error in the main body or a key function. A failed build scores 0 on the program metrics, F-score, and the judge rating; Chamfer distance is computed over successful builds. During evolution, the outer loops receive only the resulting scores; the evaluation program, its per-task specifications, and the annotations added for scoring are hidden from all agents.

E.3 Surface Metrics

F-score (Tatarchenko et al., 2019) and Chamfer distance compare full 3D surfaces over the four public articulation states after one rigid alignment and a valid joint mapping. Each model and state is sampled with 50,000 points by area. The F-score threshold is 1% of the diagonal of the initial reference bounding box, and Chamfer distance is normalized by the same diagonal. F-score varies little across methods, possibly because the task asks for a plausible articulated design rather than a reconstruction of the reference surface, and the threshold is strict. Chamfer distance orders the methods in the same way as the program score (Table 8).

E.4 Agent Judge

On the final test only, an agent judge (Zhuge et al., 2024) based on GPT-6 Astra rates holistic shape, key details, and functional plausibility on a 0–5 scale. It receives the reference data and the generated artifacts and checks their correspondence, rather than rating renderings alone. The judge is not used during evolution or for candidate selection. Its rubric (Table 6) rates whether the model works as the target does. A severe error in structure, assembly, or motion caps the score at 2; such errors include a missing main part, a functional cavity that is filled solid, main parts that are disconnected or attached to the wrong side, a key joint that is missing or of the wrong type, and a motion that runs in the reverse direction. The boundary between 3 and 4 is whether the core function is complete.

To check the judge, two human raters scored 50 generated results with the same rubric and the same inputs, independently of the judge and of each other (Table 7). The judge agrees with the mean human rating at a Spearman correlation of 0.83 and a Pearson correlation of 0.88, with a mean absolute difference of 0.41 points, and its agreement with each rater is close to the agreement between the raters. The judge is slightly more lenient, with a mean rating of 3.80 against 3.58 and 3.64. We read this as agreement at about the level of two human raters.

E.5 Harnesses and Baselines

The minimal harness is the starting point of Section 3: the Pi harness, in which the Actor can call FreeCAD, without skills, tools, or examples. The environment-informed harness is written by hand, independently of the content that evolution produces: a skill on using the FreeCAD API,

Table 6: Rubric of the agent judge (summary).

Score	Criterion
0	No checkable artifact, or the artifact is unrelated to the target.
1	Some cues of the target, but no coherent main structure or assembly; the main use is not achievable.
2	The target is recognizable, but structure, assembly, or motion has at least one severe error.
3	The main body and some main functions work without severe errors, but auxiliary functions are missing or fail, or several deviations leave the model clearly incomplete.
4	The core functions are complete and the key poses are reachable; local deviations in shape, mounting, or secondary motion are allowed.
5	Required parts, necessary cavities, assembly, and required motions are correct, with no unreasonable interpenetration during the expected motion; only decorative details differ.

Table 7: Agreement between the agent judge and two human raters on 50 results. κ_w: quadratic-weighted kappa.

Pair	Spearman	Pearson	κ_w
Judge vs. rater 1	0.83	0.85	0.81
Judge vs. rater 2	0.81	0.86	0.85
Rater 1 vs. rater 2	0.90	0.86	0.84
Judge vs. mean of raters	0.83	0.88	–

an introduction to the environment, and reference FreeCAD examples, in 8,186 characters in total. It also provides five tools, absent from the minimal harness, which write the model, build it in FreeCAD, inspect the artifacts, render the model from different viewpoints and articulation states, and check the final artifact. It contains no modeling strategies for particular objects or mechanisms.

All evolving methods start from the environment-informed harness and share the protocol of Section 5.1: GPT-5.6 Sol drives the outer loop, GPT-5.6 Luna is the Actor, and training tasks arrive in the same batches of 10 over 20 rounds with the same validation-based acceptance: a candidate harness is accepted when it improves validation performance, and the final accepted state is frozen for testing. The outer loops of all evolving methods receive the same materials, including the training references. The methods differ only in how the outer loop revises the harness. GEPA evolves the prompts of the harness with its reflective prompt evolution (Agrawal et al., 2026). For Meta-Harness, we use the outer-loop optimizer of its released implementation, with the search space of the original paper, to optimize our harness (Lee et al., 2026). In conventional evolution, a single Improver reflects on trajectories and edits the harness using our infrastructure; it may inspect the references on its own, but it has no dedicated Verifier, hypotheses, targeted trials, or localization, and it does not maintain explicit conditions and realizations. It is the loop analyzed in Section 3, and the comparison with it reflects the evidence-guided design. LAM (Gao et al., 2026) keeps its own pipeline and code representation. Its original input is a short text prompt, which we replace with the input of our task, and its output, like ours, is an articulated model with explicit parts and joints, so the same evaluation program applies.

E.6 Models and Budgets

The Actor and the Router use GPT-5.6 Luna, the outer-loop agents use GPT-5.6 Sol, task descriptions and requirements are written by GPT-5.5, and the judge uses GPT-6 Astra. A single Actor execution is limited to 0.5M tokens, including reading and writing, and stops when it reaches the limit. With our harness, the average Actor execution uses about 60K tokens at the start of evolution and about 0.2M tokens with the final harness, since the evolved experience adds context and tool calls. Each outer-loop round of every evolving method is limited to one hour of wall-clock time. Together with the shared models, batches, rounds, and acceptance rule, these limits apply equally to all evolving methods.

F On Evaluating Articulated CAD Generation

CAD generation is often evaluated with reconstruction metrics such as Chamfer distance, F-score, or volumetric IoU against a reference shape (Wu et al., 2021; Khan et al., 2024; Rukhovich et al., 2024; Tatarchenko et al., 2019). These metrics fit tasks that aim to recover a given surface. Our task is closer to generation: four views and a description do not determine every dimension, and a design can have the same parts, joints, and function as the reference while differing in geometric detail. Surface agreement therefore measures part of what matters, which is why we report F-score as a secondary metric and Chamfer distance only in Table 8.

The field lacks an agreed way to judge whether an articulated design is good. A common practice is to ask a large model to judge renderings or code (Zheng et al., 2023; Alrashedy et al., 2025), and agent judges extend this by inspecting artifacts before rating (Zhuge et al., 2024). We use two complementary approaches. The program metrics check requirements annotated from the references, covering structure, motion, and assembly. The agent judge compares the generated artifacts with the reference data. Both have limits. The checks depend on annotation and cannot capture every aspect of a design; the judge depends on the model and on how it is prompted. In our results the two rank the methods almost identically; the only swap is between Meta-Harness and conventional evolution, which are close under both (Table 1). Building evaluations for articulated CAD that are reliable across geometry, articulation, and function remains an open problem.

G Additional Results

G.1 Chamfer Distance

Table 8 adds Chamfer distance to Table 1.

Table 8: Test results with Chamfer distance (CD, lower is better).

Method	Score	CD↓
Minimal harness	0.524	0.0280
Env.-informed harness	0.626	0.0257
GEPA	0.632	0.0256
Meta-Harness	0.674	0.0231
Conventional evolution	0.683	0.0213
LAM (hand-designed)	0.666	0.0245
Evidence-guided (ours)	0.743	0.0211

G.2 What the Harness Learns

We audit the updates of our method with the categories of Figure 1(c). Over the 20 rounds, the outer loop proposed 83 updates, of which 26 were accepted. By function, none of the accepted updates is environment knowledge, 2 are mixed, and 24 are modeling experience; this is expected, since our run starts from the environment-informed harness. By realization, 12 are model-readable and 14 are executable. Conventional evolution, audited in Section 3 from the minimal harness, has 7 executable updates among 37. The two runs start from different harnesses, so we compare only the share of executable updates, which rises from about one fifth to more than half. The rollback ablation (Table 4) tests the role of executable realizations directly.

G.3 Reliability of Diagnosis Labels

Diagnosis labels are produced by the Improver and Verifier, so we checked them by hand. We randomly sampled 100 uses of experiences from the evolution, including uses that received a label and uses that were left unlabeled, and judged each from the same evidence. In 83 cases our judgment matched the system's, in 11 we found the case too entangled to attribute, and in 6 we disagreed. The agreement suggests that the labels are informative enough to guide revisions, but not that they are always correct; attributing a failure is itself partly subjective, and a human judgment is not a ground truth. Revisions are also subject to validation before acceptance.

H Discussion of Design Choices

Do the gains come from a larger budget? All evolving methods use the same models, batches, rounds, per-round time limit, and acceptance rule, and all Actor executions share the same token limit. Our outer loop spends more of each round on investigation and trials, but not more rounds or wall-clock time.

Does access to references leak test information? The Verifier checks only training artifacts against training references. The archive indexes only training executions, the harness is frozen before testing, and the evaluation program and its annotations are hidden from all agents. The references are meshes and articulation specifications, not FreeCAD programs, so there is no construction code to copy (Appendix D.3). The outer loops of all evolving methods receive the same materials, so access to references does not distinguish our method from the baselines.

Is the environment-informed starting point favorable to our method? The environment-informed harness is written by hand, independently of what evolution produces, contains no object-specific modeling strategies, and is shared by all evolving methods. Figure 1(b) shows conventional evolution from the minimal harness as well. Starting from the environment-informed harness makes the comparison stricter rather than easier, since it removes the early environment gains.

Is conventional evolution a fair baseline? It is the loop whose behavior motivates this work (Section 3), and it uses our infrastructure, so the comparison reflects the evidence-guided design rather than differences in engineering.

Could the base models have seen the data? Artiverse was released in 2026, and we cannot rule out that some of its objects or related assets appear in the training data of the base models. Our task descriptions and requirements are newly written, and all methods share the same base models.