LEARNING META-SKILLS FOR AGENT HARNESS DESIGN IN TEST-TIME AI4AI

Anonymous authors
Paper under double-blind review

ABSTRACT

Agent performance depends on both reasoning ability and the environment in which it acts. We study test-time AI-for-AI, asking how a ***Builder*** can learn to construct better execution environments for a ***Target*** while both models' weights remain fixed. To make the Builder's experience reusable, we introduce ***meta-skills***: principles specifying when support is needed and what resources to provide. The Builder learns these principles from Target's execution feedback on the development set, then uses the frozen skill bank to construct harnesses for unseen tasks. Across Harness-Bench and NewtonBench, full-bank meta-skills improve macro-average performance by 8.95 percentage points over no-skill construction, and 12.02 points over direct delivery of the same bank to the Target. These results highlight the value of translating experience into executable support. Gains when the same model serves both roles further suggest a path to system level self-improvement through learning to build better environments.

1 INTRODUCTION

An AI agent's performance depends on both its reasoning ability and the environment in which it acts (Yang et al., 2024). Consider a capable PhD student who spends days repeating experiments because configurations and results are scattered across scripts and notes. An effective advisor can address this bottleneck by establishing a shared experiment log, reproducible tools, and a clear validation workflow, helping the student devote more effort to scientific judgment. This analogy suggests a complementary direction for improving AI agents: **learning how to provide the support that makes their existing capabilities more effective** (Lee et al., 2026; Ye et al., 2026).

AI-for-AI (AI4AI) at test-time offers a route to this goal by enabling AI systems to design agent programs, workflows, and harnesses (Hu et al., 2025; Zhang et al., 2025; Lee et al., 2026). This form of AI4AI involves two complementary roles: a **Builder**, which designs and provides support, and a **Target**, which uses that support to solve tasks. We study this relationship through harness construction, where the Builder creates a *harness* comprising instructions, resources, and executable mechanisms for the Target. Just as an advisor learns from a PhD student's progress to refine their guidance, the Builder can learn from the Target's execution to improve its support. Building on recent work on meta-level learning (Ye et al., 2026), we therefore ask: **how can the Builder turn the outcomes of its own harnesses into reusable knowledge for supporting future tasks?**

To make this experience reusable, we distinguish the knowledge needed by the **Builder** from task skills used by the **Target**. Task skills describe how to perform a task (Wei et al., 2026); the Builder instead needs principles for designing the support that helps the Target perform it. We call these reusable support principles **meta-skills**. Empirically, each meta-skill shuold specify *when* support is needed, what capability or resource to *provide*, and how the Target should *use* it while retaining responsibility for judgment.

In this paper, we connect the learning and implementation of these principles through a construction–execution–reflection loop. Starting with an empty meta-skill bank, the **Builder** constructs harnesses for development tasks, reviews the resulting Target execution records and scores, and adds or revises principles based on the observed evidence. After skill learning, the bank is frozen and guides fresh harness construction for each held-out task. Learning thus changes the Builder's external knowledge and the support it constructs, while both the Builder's and Target's model weights remain fixed.

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.

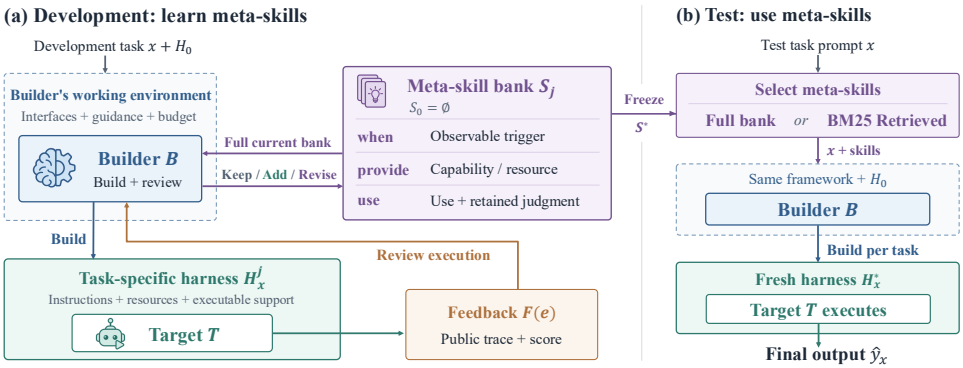


Figure 1: Overview of our framework. Rather than teaching the **Target** directly, our framework equips the **Builder** with **meta-skills** learned from Target execution feedback. These reusable principles guide the Builder in constructing harnesses for new test tasks.

We evaluate three Targets on Harness-Bench and NewtonBench under fixed Target execution budgets. The full-bank meta-skill Builder achieves a 65.31% macro-average score, exceeding the no-skill Builder by 8.95 percentage points and full-bank independently learned Target skills by 10.93 points. It also outperforms direct delivery of the same bank to the Target in all six settings. These comparisons suggest that meta-skills can help the Builder become a more effective advisor: they guide support design, while Target skills guide task execution. In our setting, teaching the **Builder** to translate experience into executable support can therefore be more beneficial than directly teaching the **Target** additional task skills.

Because **meta-skills** encode reusable principles, we further examine how they develop and whether they remain useful beyond the setting in which they were learned. Through analysis, we discover that repeated reflection can strengthen the guidance, and transfer studies suggest possible meta-skill reuse across Builders and Targets. The **Builder** and **Target** can also be instances of the same model: across three such settings, meta-skills improve scores by 18.71 points on average over no-skill construction. These results suggest that a model can improve its own execution by learning to build better support for itself, establishing harness design as a promising route to system-level self-improvement.

Looking ahead, AI4AI broadens the goal of agent learning: agents can learn both to solve problems and to create the conditions for others to succeed. Improving how Builders provide guidance and resources may be as consequential as improving how Targets use them. **Better agents may begin with better advisors.**

2 Related Work

AI for AI in Agentic Systems. AI-for-AI uses AI systems to automate model development and agent design. For model development, AIDE uses an LLM agent to write and refine machine-learning pipelines, while MLE-Dojo provides environments for training and evaluating agents performing this engineering work (Jiang et al., 2025; Qiang et al., 2026). For test-time agent design, the optimization target becomes the system surrounding a fixed model: Automated Design of Agentic Systems and AFlow both focus on searching agent programs and workflows (Hu et al., 2025; Zhang et al., 2025), while Darwin Godel Machine evolves its own agent code, and Meta-Harness searches harness implementations using execution feedback (Zhang et al., 2026a; Lee et al., 2026). Within this direction, strong-to-weak harness construction uses a stronger Builder to supply executable support for a weaker Target (Qian et al., 2026). We build on this setting by learning reusable construction principles from Target executions during offline skill learning, then freezing them to guide fresh harness construction for each test task, with both models' weights fixed.

Agent Skills and Meta-Skills. Agent skills preserve procedural knowledge for reuse: Voyager stores executable routines, while ExpeL and Agentic Context Engineering accumulate textual insights and contextual playbooks (Wang et al., 2023; Zhao et al., 2024; Zhang et al., 2026b). Methods for improving these resources include SkillRL, which co-evolves skills and policies through rein-

Table 1: Overview of the harness refinement space, including neutral defaults in H_0, Builder-editable mechanisms, and illustrative examples. The underlying benchmark tools, evaluation budget, and scoring rules remain fixed.

	Instructions	**Memory**	**Context**	**Composed tools**	**Execution control**	**Verification & recovery**	**Workspace**
Initial H_0	Generic prompt	Empty store	Default history	Native tools	Default loop	Native rules	Empty scratch
Builder $\mathcal{B}$ **refines what?**	Specific task guidance	Record format; storage and retrieval rules	History-selection rules	Tool definitions; call sequences	Execution phases; tool visibility	Submission checks; failure handling	Initial files; setup actions
Example Implement	"Draft before polishing"	Recent trial buffer	Budgeted history window	Cross-file auditor	Hide failing tools	Audit-gated submission	Templates; validator scripts

forcement learning, and Evo-Harness, which compiles execution experience into transferable solver skills (Xia et al., 2026; Wei et al., 2026). SkillsBench complements these methods by measuring skill utility with curated packages and deterministic verifiers (Li et al., 2026). At the meta level, learned knowledge guides agent improvement itself: Meta Context Engineering learns skills for constructing context files and code, while MetaSkill-Evolve evolves policies for improving task skills (Ye et al., 2026; Wang et al., 2026). Within this direction, our **Builder** learns those principles from its own harness outcomes and implements them as task-specific environments for a separate **Target**, specifying when support is needed, what capabilities to supply, and the Target's responsibilities.

3 Problem Setting and Method

Problem Formulation. Let $\mathcal{B}$ and $\mathcal{T}$ be fixed **Builder** and **Target** models. Each benchmark provides a baseline environment H_0 with native capabilities, an evaluator r, and a Target execution budget C_x. A harness policy $\mathcal{H}$ maps each public task input x to an environment $H_x = \mathcal{H}(x)$ that augments H_0 with executable support. The objective is to maximize expected test performance:

$$J(\mathcal{H}) = \mathbb{E}_{x\sim\mathcal{D}_{\text{test}},,\tau\sim\mathcal{T}(\cdot|x,H_x)}[r(x,\tau)], \qquad \text{cost}(\tau) \le C_x. \tag{1}$$

The **Builder** learns the policy through Target executions on development instances and is evaluated on held-out test instances. The budget C_x covers only Target execution, excluding Builder computation for harness construction.

Method Overview. Instead of directly improving the **Target**, we teach the **Builder** to learn what support the Target needs and provide it through a harness. The Builder's experience accumulates in an external bank of **meta-skills**, which records reusable principles for designing support across tasks. These principles guide the harness construction, while both the Builder and Target model weights remain fixed throughout learning.

3.1 Learning Meta-Skills from Target Execution During Development

Definition of Meta-Skill. A **meta-skill** $s = (\text{when}, \text{provide}, \text{use})$ contains three fields: *when* identifies observable conditions that call for support; *provide* specifies the capability or resource the environment should supply; and *use* explains how the **Target** should employ that support and which judgments remain its responsibility.

Skill Learning Workflow. Skill learning begins with an empty skill bank S_0. For each development task input x, the **Builder** constructs a separate harness using the neutral environment H_0, public component interfaces, and its complete current bank. The **Target** executes within the harness, producing a public execution record and a benchmark-native development score. The same Builder then updates its skill bank by reviewing the current bank, its generated programs, and public execution feedback $F(e)$. This process will iterate until a fixed budget of development-set passes is reached, as formalized below:

$$H_x^j = \mathcal{B}(H_0, x, S_j), \qquad e_x^j = \mathcal{T}(x; H_x^j), \qquad S_{j+1} = \text{Revise}_{\mathcal{B}}\big(S_j, \{(H_x^j, F(e_x^j))\}_{x\in G_j}\big). \tag{2}$$

Skill Bank Update. At the end of each skill-learning step, the **Builder** reviews its generated harnesses and the Target's execution feedback to identify an observed error or recurring burden that

better support could address, then compares the resulting lesson with the current bank. It *revises* an existing meta-skill when the evidence corrects or strengthens its guidance, *adds* a new meta-skill for a distinct reusable support need, or *keeps* the bank unchanged when no update is justified. Each batch permits at most one addition or revision, which must cite supporting evidence from that batch to keep the learned guidance grounded in observed behavior.

3.2 Building New Harness for Every Task at Test-Time

Test-Time Skill Selection. After skill learning, we freeze the meta-skill bank and use it to guide test-time harness construction. For each test task, the Builder receives skills through one of two modes: *full bank* places all meta-skills in its context; *retrieval* uses a fixed BM25 retriever to select at most two skills with positive relevance scores against the task's initial public prompt. For each individual test task, the **Builder** uses the supplied meta-skills to construct a new, task-specific harness in a fresh environment for the **Target** to execute within:

$$S^* = S_J, \qquad H_x^* = \mathcal{B}(H_0, x, K(x, S^*)), \qquad \hat{y}_x = \mathcal{T}(x; H_x^*), \tag{3}$$

where K supplies either retrieved meta-skills or the full bank.

Harness Components. As summarized in Table 1, the framework exposes seven optional harness component families: instructions, memory, context organization, composed tools, execution control, verification and recovery, and workspace preparation. The **Builder** selects and implements these components, deciding what memory stores, when it is retrieved, and which tools and resources the Target receives. Within the resulting harness, the **Target** reasons over observations, uses the available tools, and remains responsible for the final submission.

Builder's Working Environment. The **Target** operates within a Builder-created harness, while the **Builder** operates within a framework we design. Specifically, we provide shared interfaces, execution isolation, design guidance, and resource limits, with identical construction permissions and bounded interface-repair opportunities across conditions. These fixed priors underpin our workflow for automated meta-skill learning, harness construction, and Target execution. Our contribution thus shifts design effort to the Builder level, enabling it to *turn execution experience into reusable support principles* and adapt their implementation to individual tasks.

Example. Consider an artifact edited after validation, making the earlier check potentially outdated. A **meta-skill's** *when* field identifies this condition; *provide* requests version tracking and revalidation; and *use* directs the **Target** to inspect the current validation result before submitting, while retaining responsibility for content judgment and correction. The **Builder** could then implement this principle with file-version memory and a validation tool. In Appendix we trace two actual test episodes from learned principles through Builder-written code and Target tool calls to final artifacts.

4 Experiments

4.1 Experimental Setup

Datasets. We evaluate the effect of test-time learned support on task performance using Harness-Bench and NewtonBench, which cover agent workflows and interactive scientific law discovery, respectively (Yao et al., 2026; Zheng et al., 2026). For each benchmark, we reserve 10% of tasks for learning and use the remainder exclusively for evaluation. See Table 2 for details.

Table 2: Benchmark statistics and splits.

Benchmark	Tasks	Development	Test
Harness-Bench	106	11	95
NewtonBench	324	32	292

Models and Settings. We use GPT-5.6-Sol as the Builder model, and evaluate Gemini-3.6-Flash, Qwen3.8-Flash, and GPT-OSS-120B as Target models. For each Builder–Target–benchmark combination, the meta-skill bank starts empty and is updated over two development-set passes, with at most one evidence-grounded *keep*, *revise*, or *add* update per task. The bank is then frozen for testing, while the Builder constructs a fresh harness for each test task.

All models use temperature zero and high reasoning effort (or the corresponding thinking mode), with a 16K output-token limit for harness generation and an 8K limit for skill update. For the

Table 3: Test performance (%) on every task in the fixed test splits with GPT-5.6-Sol as Builder in all settings. Bold marks the best result in each model–benchmark column.

Method	Harness-Bench			NewtonBench			Macro Avg.
	Gemini	Qwen	GPT-OSS	Gemini	Qwen	GPT-OSS	
Native environment	37.35	69.63	52.46	55.48	54.11	39.38	51.40
Builder, no skills	53.29	71.51	63.02	56.16	50.34	43.84	56.36
Direct Builder skills, retrieved	46.66	68.98	59.13	58.22	48.29	33.22	52.42
Direct Builder skills, all	42.38	68.70	58.34	58.56	55.82	35.96	53.29
Independent Structured Target skills, retrieved	41.95	72.51	45.21	67.81	53.42	36.30	52.87
Independent Structured Target skills, all	40.59	72.89	56.63	67.81	52.40	35.96	54.38
Mined Free-form Target skills, retrieved	39.61	63.51	**71.12**	46.92	54.45	33.90	51.59
Mined Free-form Target skills, all	40.70	66.10	66.01	41.44	54.11	39.04	51.23
Builder meta-skills, retrieved	61.50	**74.81**	64.74	64.04	57.88	39.73	60.45
Builder meta-skills, all	**67.81**	72.64	68.22	**68.84**	**63.70**	**50.68**	**65.31**

benchmark budget, Harness-Bench allows 30 model turns, 30 tool calls, and 96K cumulative tokens per task; NewtonBench allows 12 turns, 10 tool calls, and 192K tokens.

Metrics. Harness-Bench reports the deterministic completion-oracle score over 95 test tasks, averaged and scaled to a percentage. NewtonBench reports symbolic-structure accuracy over 292 test tasks. Both use end-to-end evaluation, divided by the fixed number of test cases: valid native outcomes are retained, while audited harness-construction failures receive a score of zero because no executable harness is produced.

Baselines. We compare five baselines. Learned guidance is supplied either in full or through BM25 top-2 retrieval.

- **Native environment:** The Target solves tasks in the shared neutral environment, without learned guidance or Builder-generated support.
- **No-skill Builder:** The Builder constructs harnesses with the same construction space and budget as our method, but with an empty skill bank.
- **Direct Builder skills:** The Target receives our Builder's meta-skills directly as instructions, with no Builder-generated harness components.
- **Independent structured Target skills:** An equally budgeted GPT-5.6-Sol learner iteratively extracts and refine structured problem-solving skills from the Target's own neutral-environment executions. The resulting bank is supplied directly to the Target.
- **Mined free-form Target skills:** GPT-5.6-Sol consolidates the Target's neutral-environment development trajectories into free-form notes in one offline pass. These notes are supplied directly to the Target.

Our **Builder meta-skill** conditions instead provide the retrieved or full bank to the Builder, which compiles it into task-specific harness components such as instructions, memory, context, tools, controllers, verification, or workspace preparation. The full-bank Direct baseline therefore controls for semantic knowledge, while No-skill Builder controls for construction capability, serving as complementary ablations of our method. Please see Section C for more setting details.

4.2 Main results

We present the main results and baselines in Table 3, and highlight the following key findings.

Meta-skills are most useful when the teacher can enact them. Giving the same full meta-skill bank to the **Builder** consistently outperforms giving it directly to the **Target**, improving every model–benchmark pair by up to 25.43 points and 12.02 on average. This gap shows that the gains come not merely from exposing the Target to better knowledge, but from *operationalizing* that knowledge before execution. The Builder can translate a declarative meta-skill into persistent state, executable tools, verification logic, or control decisions, reducing the burden on the Target to interpret and apply the advice correctly at inference time. **Meta-skills** therefore function less as solver prompts and more as a compact language for *provisioning* task-specific support.

Experience improves the Builder beyond construction capability. With construction capability held fixed, the full-bank Builder outperforms the no-skill Builder in all six settings, with an average gain of 8.95 points. These results suggest that experience adds value beyond the ability to construct

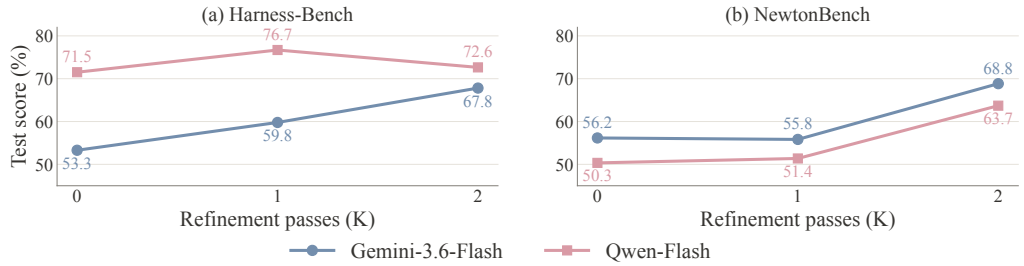


Figure 2: Test performance as Builder experience is refined. The two NewtonBench curves improve mainly after the second pass; Harness-Bench displays both steady improvement and late regression.

scaffolds: it helps the **Builder** decide *which* support to build and *how* to integrate it with Target behavior. The gains nevertheless vary across Targets and benchmarks, suggesting that accumulated guidance must be adapted to the Target and task rather than assumed uniformly useful.

Experience helps most when coordination is the main bottleneck. Relative to the no-skill Builder, gains on NewtonBench are positive across all three Targets and average 10.96 points, versus 6.95 points on Harness-Bench. NewtonBench requires agents to coordinate experimentation, reasoning, and valid symbolic submission, creating *recurring coordination failures* that scaffolding can address. Harness-Bench spans more heterogeneous workflows, where effective support depends more on *Target-specific planning and tool use*. This contrast suggests that accumulated experience is most useful when failure modes recur consistently across tasks and Targets.

Broad skill access usually outperforms sparse retrieval. The full skill bank outperforms top-2 retrieval in five of six settings, with an average gain of 7.19 points on NewtonBench. This pattern suggests that a compact meta-skill bank offers *complementary procedures* whose combined value may be missed by lexical retrieval. Full-bank access allows the **Builder** to consider these procedures together when designing support, making it a strong default at the evaluated scale. The exception, however, indicates that broader access is not uniformly beneficial and motivates retrieval methods that account for both skill complementarity and the Target's likely response to support.

5 Analysis

5.1 Effect of Meta-skill Refinement

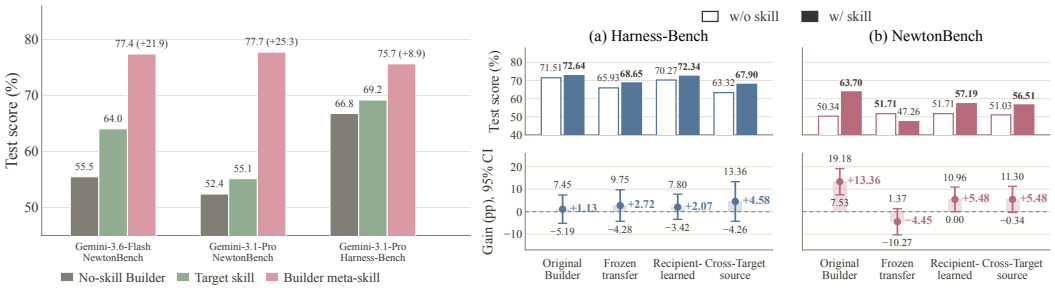
Motivation and setting. Meta-skills develop through repeated Builder reflection, but additional updates need not improve teaching. To examine this progression, we freeze the skill bank after zero, one, or two complete development passes and evaluate each version on the full test split. Figure 2 reports results for both Gemini and Qwen Targets.

Useful teaching behavior can emerge late. On NewtonBench, the first pass changes each Target's score by less than 1.1 points, whereas the second adds 13.01 points for Gemini and 12.33 for Qwen. Across all settings, the average gain from zero to two passes is 10.42 points. Manual inspection suggests that early update gathers local observations, while later revision connects them into *reusable interventions*. This delayed improvement is consistent with the value of iterative revision. The lower Mined Target skill score also motivates going beyond offline trajectory summarization, although that comparison changes both the learning procedure and skill recipient.

Refinement is not monotonic. On Harness-Bench, Gemini improves with each pass, but Qwen drops 4.06 points from its first-pass peak. Revisions can therefore sharpen useful guidance while also making it *overly specific to recently observed evidence*. Together, these results suggest that effective learning requires both continued refinement and selective retention. A practical update rule should use development-only evidence to assess confidence in revisions and decide whether to continue, retain an earlier version, or roll back a change.

5.2 Self-Improvement through Harness Design

Motivation and setting. Meta-skills need not rely on a stronger external teacher: the same model can serve as both **Builder** and **Target**. We evaluate three settings using Gemini-3.6-Flash and



(a) Same-model self-evolution **(b) Cross-Builder transfer**

Figure 3: **(a)** Same-model self-evolution compares no-skill construction, independent Target skills, and Builder meta-skills. Meta-skill bar labels show absolute scores and gains over no-skill construction. **(b)** Cross-Builder transfer on Harness-Bench and NewtonBench. Top: test scores without (outlined bars) and with (filled bars) Builder meta-skills. Bottom: paired gains in percentage points with 95% task-bootstrap confidence intervals; dashed lines indicate zero gain.

Gemini-3.1-Pro, with construction, reflection, and execution performed by the same model within each case. This setting tests whether a model can use past execution experience to improve its own future performance through *learned support design*.

Support design is a distinct target for self-improvement. Across the three settings in Figure 3(a), Builder meta-skills yield average gains of 18.71 points over no-skill construction and 14.14 points over skills delivered directly to the Target. These results identify an additional axis of optimization: improving how a model equips itself, without weight updates or a stronger teacher. This mechanism could complement the Target's own skill learning: as the Target's capabilities improve, the Builder could adapt its support to the Target's changing needs, while execution feedback informs further support refinement. Self-evolution could therefore involve coordinated improvement in both task-solving capabilities and the environments that support them.

5.3 Meta-Skill Transfer across Builders and Targets

Motivation and setting. **Meta-skills** specify support principles while leaving implementation to the **Builder**. We test whether a learned bank remains useful when another Builder translates it into harnesses. All evaluations use Qwen-Flash as Target on the complete test splits of Harness-Bench and NewtonBench, comparing four settings:

- **Original Builder reuse:** GPT-5.6-Sol learns from Qwen-Flash's development executions and uses its own bank to construct test harnesses.
- **Transfer across Builders:** Gemini-3.1-Pro constructs harnesses using the frozen Sol-to-Qwen bank, changing the Builder while holding the bank and Target fixed.
- **Recipient Builder learning:** Gemini-3.1-Pro learns its own bank from Qwen-Flash's development executions and uses it to construct test harnesses.
- **Transfer across Builders and Targets:** Gemini-3.1-Pro receives a frozen bank learned by Sol from Gemini-3.6-Flash's executions and constructs harnesses for Qwen-Flash, changing both the Builder and the Target relative to skill acquisition.

Banks are learned on each benchmark's development split and frozen before testing. Each receiving Builder uses the full bank to construct a fresh harness per test task. Figure 3(b) reports scores and paired differences against contemporaneous no-skill runs of the same Builder.

Meta-skills support cross-Builder reuse, with benefits dependent on implementation. On NewtonBench, the Sol-to-Qwen bank yields a 13.36-point gain with Sol but a 4.45-point decline when transferred to Gemini-Pro; on Harness-Bench, the same transfer instead yields a 2.72-point gain. These results support the feasibility of cross-Builder reuse, although all confidence intervals include zero, leaving the performance benefits uncertain. The differing outcomes suggest that transfer depends partly on *how the receiving **Builder** translates shared lessons into a harness*: even the skill bank is fixed, different Builders may implement the same guidance differently. Refining the resulting harness through execution feedback may therefore help realize the value of transferred meta-skills.

Table 4: Harness component ablations on NewtonBench. We report test scores (%), changes (ablation minus full) using the main-table full-bank execution as the reference, and unadjusted 95% paired task-bootstrap confidence intervals for these changes. Intervals containing zero indicate uncertainty in the direction of the average effect.

Target model	Full Harness	w/o Controller	95% CI	w/o Memory + Context	95% CI
Gemini-3.6-Flash	68.84	$55.48^{-13.36}$	$[-18.49, -8.22]$	$66.78^{-2.05}$	$[-7.53, 3.08]$
Qwen-Flash	63.70	$58.90^{-4.79}$	$[-10.27, 0.68]$	$59.59^{-4.11}$	$[-9.25, 1.03]$
GPT-OSS-120B	50.68	$49.66^{-1.03}$	$[-6.51, 4.11]$	$54.11^{+3.42}$	$[-2.05, 8.90]$

Recipient-side learning yields stronger observed performance. Gemini-Pro's own bank outperforms the imported Sol-to-Qwen bank by 3.69 points on Harness-Bench and 9.93 points on NewtonBench, with positive gains over its no-skill controls on both benchmarks. This pattern suggests that *learning from the recipient's own harness executions still better align meta-skills with its implementation choices*. Recipient-side learning may therefore complement cross-Builder reuse: imported skills provide reusable guidance, while feedback from the recipient's harnesses offers a basis for refining that guidance.

Useful support principles may transfer across both Builders and Targets. Transferring Sol's Gemini-derived bank to Gemini-Pro building harnesses for Qwen yields gains of 4.58 points on Harness-Bench and 5.48 points on NewtonBench. On NewtonBench, the transferred bank scores only 0.68 points below Gemini-Pro's own Qwen-derived bank and even outperforms the imported Sol-to-Qwen bank, despite the latter matching the receiving Target. This pattern suggests that *the relevance of learned support principles may matter more than an exact source-Target match*: guidance learned for one Builder–Target pair can remain useful when another **Builder** implements it for a different **Target**.

5.4 Harness Component Ablations

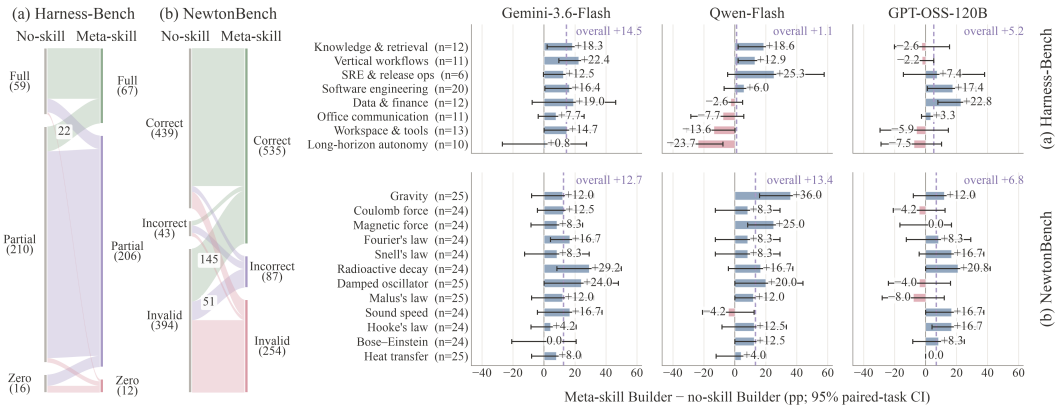
Motivation and setting. Meta-skills affect execution through the harnesses the **Builder** creates. Using NewtonBench as a case study, we freeze the constructed harnesses and replay every test task after removing either the execution controller or memory plus context. Table 4 reports the average score difference between complete and ablated harnesses, with negative values favoring retention. The accompanying *confidence intervals* quantify uncertainty in this average effect across test tasks.

Controllers generally improve harness performance across Targets. Removing the controller lowers scores for all three Targets: by 13.36 points for Gemini, 4.79 for Qwen, and 1.03 for GPT-OSS. These consistent observed drops support the controller's role in improving execution, although only Gemini's confidence interval excludes zero. The size of the benefit varies with both *Target behavior and controller design*: each ablation isolates the controller within a frozen harness, but its functions may differ across Targets, including experiment sequencing, state tracking, or submission management. Effective harness design therefore uses external coordination where it helps, tailoring the controller's functions to each Target's execution needs.

Memory and context show promise as targeted support. Retaining memory plus context yields small positive estimated effects for Gemini and Qwen and a negative effect for GPT-OSS, although all three confidence intervals contain zero. These estimates suggest potential benefits for some Targets, but do not establish a reliable aggregate effect. Additional state can preserve useful evidence, while its value depends on keeping that evidence relevant and avoiding redundant or outdated information. Meta-skills therefore should support *selective state exposure* by guiding what to retain and when to surface it for the **Target**.

5.5 Outcome Transitions and Error Analysis

Motivation and setting. Aggregate scores can conceal both changes in execution outcomes and offsetting gains and regressions across task families. We pair the full-bank meta-skill and no-skill Builders on all model–task pairs across the six main-study settings. In Figure 4, panel (a) aggregates outcome transitions across the three Targets within each benchmark, showing which outcomes are rescued or degraded, while panel (b) breaks down paired score differences by Harness-Bench task



(a) Outcome transitions **(b) Category-level effects**

Figure 4: **(a)** Outcome transitions from no-skill to full-bank meta-skill Builders, aggregated over three Targets. Band widths represent paired-task counts; in-band numbers highlight the largest cross-outcome flows. "Zero" includes zero scores and construction failures; "Invalid" includes missing or unparsable submissions and construction failures. **(b)** Category-level effects across all six settings. Rows represent Harness-Bench task categories and NewtonBench physical mechanisms; columns represent Targets. Bars show score differences between full-bank and no-skill Builders with unadjusted 95% task-bootstrap confidence intervals. Purple dashed lines indicate setting's overall effect.

category and NewtonBench physical mechanism for each Target, revealing where improvements concentrate and where failures persist.

Meta-skills help close the execution gap. On NewtonBench, cases with no valid submission fall from 394 to 254 (35.5%), while correct discoveries rise from 439 to 535 (21.9%). Transitions include 145 recoveries from no valid submission to a correct law, versus 46 regressions. This suggests more reliable completion of the scientific loop, creating opportunities for correct discovery. Valid-but-incorrect submissions also rise from 43 to 87, showing that completion enables but does not guarantee correctness. On Harness-Bench, full successes increase from 59 to 67, largely through transitions from partial to full success. This suggests that meta-skills help agents satisfy remaining requirements and *turn partial progress into complete task fulfillment*.

Meta-skill benefits recur across Targets, with task-dependent variation. All three Targets improve overall on NewtonBench by 6.85–13.36 points, with positive category-level point estimates for gravity, radioactive decay, and Hooke's law across models. Harness-Bench shows gains for every Target on SRE/release tasks, with mixed effects elsewhere. GPT-OSS improves by 5.20 points overall but declines by 2.23 points on vertical workflows, illustrating that a positive aggregate gain can coexist with category-specific regressions. These recurring gains suggest shared support needs across models, while the remaining variation points to the importance of Target–task compatibility. This further motivates *reusing support principles while adapting their implementation* to each Target's capabilities and the task's demands.

6 Conclusion

In this work, we introduced **meta-skill** that enable a **Builder** to learn reusable support principles from the execution outcomes of its own harnesses and implement them for unseen tasks. With both models' weights fixed, full-bank meta-skills improve macro-average performance across Harness-Bench and NewtonBench by 8.95 percentage points over no-skill construction and 12.02 points over direct delivery of the same bank to the Target. These comparisons highlight the value of translating experience into executable support: a lesson can guide the construction of tools, state, and control mechanisms that help the Target apply its capabilities during execution. Outcome analyses connect these gains to more reliable task completion, while component and transfer results point to the importance of adapting reusable principles to each model and task. Same-model experiments further demonstrate that a model can self-improve by learning to build better support for itself. Together, these findings identify support design as a complementary axis of agent improvement.

Ethics Statement

This work studies automated harness design using existing agent-workflow and simulated scientific-discovery benchmarks. Experiments use isolated execution environments and bounded resources, with development and test tasks separated and meta-skill banks frozen before evaluation to preserve evaluation integrity. Because generated tools and controllers can enable unintended or harmful actions, applications beyond these benchmarks should incorporate restricted permissions, validation of generated components, and human oversight appropriate to the application.

Reproducibility Statement

Section 3 describes the meta-skill representation, learning and update procedures, harness design space, and test-time construction protocol. Section 4.1 and Table 2 document the benchmark split sizes, model configurations, learning and execution budgets, baseline definitions, and scoring conventions, including the treatment of construction failures. Sections 5.1–5.4 describe the refinement, same-model, transfer, and component-ablation protocols, while Section 5.5 explains the paired outcome and category-level analyses. Together, these descriptions support reproduction of the learning, construction, and evaluation pipeline and comparison under consistent experimental conditions.

AI Use Statement

In this work, LLMs are used strictly for research support rather than as sources of substantive content. Their use falls into three categories: (i) providing tested results on our AI4AI at test-time framework, and (ii) assisting with language refinement during paper writing. For writing support, we used ChatGPT solely to polish text (improving coherence and grammar) while all ideas, logic, results, and technical contributions originate from the authors. To safeguard rigor, we have carefully reviewed all LLM-refined texts to confirm that no hallucinated content was introduced and that the original arguments, findings, and perspectives were faithfully preserved.

References

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *International Conference on Learning Representations*, volume 2025, pp. 21344–21377, 2025.

Zhengyao Jiang, Dominik Schmidt, Dhruv Srikanth, Dixing Xu, Ian Kaplan, Deniss Jacenko, and Yuxiang Wu. Aide: Ai-driven exploration in the space of code. *arXiv preprint arXiv:2502.13138*, 2025.

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.

Xiangyi Li, Yimin Liu, Wenbo Chen, Bingran You, Zonglin Di, Yifeng He, Shenghan Zheng, Kyoung Whan Choe, Jiankai Sun, Shuyi Wang, et al. Skillsbench: Benchmarking how well agent skills work across diverse tasks. *arXiv preprint arXiv:2602.12670*, 2026.

Cheng Qian, Wenting Zhao, Liangwei Yang, Heng Wang, Jielin Qiu, Heng Ji, Silvio Savarese, Huan Wang, and Shelby Heinecke. Ai4ai at test-time: Strong-to-weak capability transfer via harnesses. *arXiv preprint arXiv:2608.12307*, 2026.

Rushi Qiang, Yuchen Zhuang, Yinghao Li, Dingu Sagar VK, Rongzhi Zhang, Changhao Li, Ian Wong, Sherry Yang, Percy Liang, Chao Zhang, et al. Mle-dojo: Interactive environments for empowering llm agents in machine learning engineering. *Advances in Neural Information Processing Systems*, 38, 2026.

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. *arXiv preprint arXiv:2305.16291*, 2023.

Zefeng Wang, Minxi Yan, Jinhe Bi, Sikuan Yan, Volker Tresp, and Yunpu Ma. Metaskill-evolve: Recursive self-improvement of llm agents via two-timescale meta-skill evolution. *arXiv preprint arXiv:2607.05297*, 2026.

Tianxin Wei, Zhan Shi, Minhua Lin, Bing He, Zewen Liu, Yisi Sang, Yuanchen Bei, Xuying Ning, Jiaru Zou, Ting-Wei Li, et al. Evo-harness: Context-to-harness skill compilation for self-evolving agents. *arXiv preprint arXiv:2608.15071*, 2026.

Peng Xia, Jianwen Chen, Hanyang Wang, Jiaqi Liu, Kaide Zeng, Yu Wang, Siwei Han, Yiyang Zhou, Xujiang Zhao, Haifeng Chen, et al. Skillrl: Evolving agents via recursive skill-augmented reinforcement learning. *arXiv preprint arXiv:2602.08234*, 2026.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

Yilun Yao, Xinyu Tan, Chao-Hsuan Liu, Yaoming Li, Zhengyang Wang, Wenhan Yu, Zhewen Tan, Yuxuan Tian, Guangxiang Zhao, Lin Sun, et al. Harness-bench: Measuring harness effects across models in realistic agent workflows. *arXiv preprint arXiv:2605.27922*, 2026.

Haoran Ye, Xuning He, Vincent Arak, Haonan Dong, and Guojie Song. Meta context engineering via agentic skill evolution. *arXiv preprint arXiv:2601.21557*, 2026.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin gödel machine: open-ended evolution of self-improving agents. In *International Conference on Learning Representations*, volume 2026, pp. 104223–104294, 2026a.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, et al. Aflow: Automating agentic workflow generation. In *International Conference on Learning Representations*, volume 2025, pp. 34040–34077, 2025.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, et al. Agentic context engineering: Evolving contexts for self-improving language models. In *International Conference on Learning Representations*, volume 2026, pp. 86069–86100, 2026b.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: Llm agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19632–19642, 2024.

Tianshi Zheng, Kiu Wai Tam, Kim Hue Nam Nguyen, Baixuan Xu, Zhaowei Wang, Cheng Jiayang, Hong Ting Tsang, Weiqi Wang, Jiaxin Bai, Tianqing Fang, et al. Newtonbench: Benchmarking generalizable scientific law discovery in llm agents. In *International Conference on Learning Representations*, volume 2026, pp. 99018–99088, 2026.

APPENDIX

A SIGNIFICANCE, SCOPE, AND CLARIFICATIONS

A.1 WHY BUILDER LEARNING MATTERS

We discuss the significance of Builder learning and highlight the core contributions of this study.

Learning to Design Useful Support. We study how a Builder can turn experience into reusable principles for supporting (or harnessing) a Target. These meta-skills specify when support is needed, what to provide, and how the Target should use it. They guide fresh harness construction on subsequent tasks, allowing the Builder to carry forward design knowledge while adapting its implementation to each task. This makes support design itself an object of learning.

Turning Experience into Executable Support. In the main study, giving the full skill bank to the Builder yields higher scores than giving the same bank directly to the Target in all six settings. This highlights the value of translating guidance into tools, state, and execution control that the Target can use. The same-model experiments further illustrate a route to system-level self-improvement: a model can use execution experience to learn how to equip itself more effectively, with its weights unchanged.

A.2 CLARIFICATIONS OF RESULTS

We explain the rationale for our experimental settings and clarify the conclusions supported by our results.

Contribution within test-time AI4AI. We study how experience improves a Builder's ability to support a Target while both models' weights remain fixed. Closely related work learns solver guidance, as in Evo-Harness (Wei et al., 2026); searches executable harness implementations, as in Meta-Harness (Lee et al., 2026); or evolves strategies for engineering context files and code, as in Meta Context Engineering (Ye et al., 2026). Our focus is reusable knowledge for support design: the Builder extracts principles from its own harness outcomes, specifying when support is needed, what to provide, and how the Target should use it. These principles subsequently guide fresh construction for held-out tasks. Our contribution combines this learning and deployment protocol with controlled evidence on the value of accumulated Builder experience and its implementation as task-specific support.

Shared framework and interpretation of the comparisons. All Builder conditions share component interfaces, design guidance, construction permissions, and bounded repair opportunities. The no-skill Builder assesses the additional value of learned guidance under the same construction allowance. Direct-bank baselines preserve the skill content and assess the benefit of having a Builder implement it before execution, while independently learned Target skills and mined notes provide alternative routes for applying experience to task solving. Full-bank macro-average gains of 8.95 points over no-skill construction and 12.02 points over direct delivery support the value of learning and implementing support principles within this framework. These comparisons fix Target execution limits; learning and construction fall outside those limits, with construction allowances matched across Builder conditions. Frozen-harness ablations examine component contributions within the resulting implementations, and case studies illustrate how principles become executable support. Their Target-dependent results, including the clear controller benefit for Gemini-3.6-Flash, support adapting available components to the recipient's needs.

Evaluation protocol and reproducibility. Skill acquisition uses development episodes, and the bank is frozen before testing. Each test harness is constructed from the public task input and the supplied bank, without access to test execution feedback or reference answers. Appendices B and C document the core settings: the partitioning procedure, information access, model choices, resource limits, baseline configurations, failure accounting, and paired task-bootstrap procedures. All declared test tasks remain in the evaluation denominator, and reported intervals characterize uncertainty across paired test instances. Upon paper acceptance, we will release the full code and all prompts for skill learning, harness construction, baseline runs, and evaluation, together with the experimental configurations.

Generalization to new tasks within each benchmark. Banks learned from approximately 10% of each benchmark guide construction on the remaining held-out instances, with full-bank improvements over no-skill construction in all six settings. This establishes the usefulness of accumulated experience for new tasks within known categories and mechanisms, covering both workflow execution and scientific discovery. Cross-dataset reuse is a natural extension beyond this evaluation scope and is left to future work. A promising hypothesis is that skill granularity is central to broader reuse: principles should specify recurring support needs precisely enough to guide construction while leaving their implementation adaptable. Separating support principles from executable realizations provides a concrete basis for studying this balance.

Transfer as reuse across Builder–Target configurations. Our transfer study examines reuse of a frozen bank when the Builder or the Builder–Target pair changes. Effects are measured against concurrent no-skill runs of the receiving Builder and interpreted through paired confidence intervals. Cross-Builder-and-Target transfer yields positive point estimates on both benchmarks, illustrating promising reuse across selected model combinations. The broader results establish the feasibility of sharing principles while showing that their realized utility depends on the receiving configuration, with performance benefits qualified by the reported uncertainty. Recipient-side learning offers a complementary route for aligning support principles with a Builder's own implementations.

A simple foundation for meta-skill evolution. We adopt a simple initial evolution mechanism to make the contribution of experience directly assessable. Starting from an empty bank, the Builder constructs harnesses, reviews Target execution, and makes one evidence-grounded decision per development task: keep, revise, or add. Two development passes produce the bank used for evaluation. Held-out gains demonstrate that useful support principles can emerge through this lightweight procedure, while same-model experiments extend the result to self-improvement without a stronger external teacher. The refinement curves show that update effects depend on the Target and task, motivating future advances in selective revision and retention on top of this working foundation.

A.3 Scope and Future Directions

We define the scope of the present study and identify promising directions for future research that build on its findings.

Support design as the object of learning. This work studies learning that changes the knowledge and execution environments surrounding fixed models. The reported improvements therefore concern the resulting Builder–Target system, including system-level self-improvement when the same model serves both roles. The meta-skill representation and component interfaces provide a concrete instantiation of this learning problem within a shared design space. Our contribution establishes the usefulness of this route under the evaluated conditions and provides a reference point for studying alternative support representations and construction policies. A complementary direction is to combine Builder learning with improvements to the Target itself, allowing task-solving capabilities and the support that makes them effective to develop together.

Empirical scope and explanatory claims. We evaluate end-to-end task performance, encompassing both benchmark correctness and the execution processes needed to achieve it. Preserving evidence, validating artifacts, and reaching a valid submission are substantive parts of this objective. The empirical effects are conditional on the evaluated task inventories, models, learned banks, and recorded executions. Component analyses examine contributions within the generated harnesses, while case studies illustrate concrete ways in which support changes execution. These findings provide an empirical basis for further work on stability and mechanism: independent learning runs, repeated executions, and alternative task partitions can characterize variability across experience histories, and targeted interventions on individual principles and component interactions can refine the causal account of how support helps.

Generality through shared support needs. The main generalization claim concerns new instances within known workflow categories and physical mechanisms; the model-transfer studies provide exploratory evidence for selected receiving configurations. Extending reuse across datasets, unseen task families, and different tool interfaces is a promising next step. One hypothesis is that the scope of a principle depends strongly on the granularity at which it describes a support need. For example, preserving evidence after a tool failure may be useful across domains even when the appropriate storage and recovery mechanisms differ. Studying this separation between a reusable need and its

local implementation could clarify when principles transfer directly, when recipient-side adaptation helps, and how to learn banks that serve broader collections of tasks.

Evolution over longer learning histories. The current two-pass development procedure followed by frozen-bank deployment demonstrates that a bounded amount of experience can improve subsequent harness construction. Extending this setting to longer task streams would allow study of how useful principles are retained, revised, and combined as experience accumulates. The observed variation between refinement passes motivates development-based assessment of candidate updates and selective retention of successful guidance. Larger banks also create opportunities for adaptive retrieval and skill composition, extending the current comparison of full-bank access and BM25 top-2 retrieval. These directions would develop the present initial mechanism into a learning process suited to longer histories and changing support needs.

Performance objectives and deployment conditions. The experiments study performance under fixed Target interaction limits, with Builder learning and construction providing support outside that execution budget. This setting makes the value of prepared support assessable for a given Target operating under the same interaction constraints. The supported performance claim uses this budget definition. For deployment, a complementary objective is to allocate a joint compute or latency budget across learning, construction, and execution according to the workload. Future work can study how reuse of learned banks, caching of common support components, and task-adaptive construction effort affect this allocation, connecting the demonstrated performance benefits to different deployment requirements.

B Reproducibility Details

B.1 Data partitions and information access

For each benchmark, we reserve a development set of approximately 10% of the instances for meta-skill learning and use the remainder for testing. We determine the development-set size by rounding ten percent of the total to the nearest integer, with ties rounded up, and allocate proportional integer quotas across the public Harness-Bench categories and NewtonBench domains. This partitioning evaluates generalization to held-out instances within known categories and mechanisms. We use no separate validation partition, and the learned skill bank is frozen before testing, with no revisions based on test outcomes.

Each meta-skill contains *when*, *provide*, and *use* fields, with a combined limit of 192 tokens. During learning, the Builder receives the entire current bank and updates it using evidence from learning episodes. The actual submission, or its absence, serves as the primary evidence, interpreted alongside the public task requirements, complete input–observation pairs, public errors, final artifacts, and support delivered by the harness. Each update produces a single decision: keep the bank unchanged, revise an existing skill, or add a new skill. A revision must identify the existing skill and the current evidence motivating the change; superseded versions remain archived. Neither private reference answers nor researcher-written diagnoses are provided during this process.

At deployment, BM25 queries the skill bank using only the initial public task prompt and retrieves at most two skills with positive match scores. Harness construction receives the public initial task, neutral component interfaces, and the assigned semantic skill pack. The Builder has no access to test trajectories, scores, reference answers, or evaluator internals, so construction depends entirely on the information available before Target execution.

B.2 Models and resource limits

We use GPT-5.6-Sol as the Builder for both harness construction and meta-skill updates. The Target models are Gemini-3.6-Flash, Qwen3.8-Flash, and GPT-OSS-120B. All requests specify a temperature of zero and request high reasoning effort.

Harness construction allows up to 16K output tokens, with at most two repair attempts for candidates that violate the component interfaces. Each skill-update call allows up to 8K output tokens and returns the single keep, revise, or add decision described above. Target execution follows the

benchmark-specific resource allowances listed below; these allowances are identical across experiment settings.

Table 5: Per-task Target limits. Each model call permits at most 16,384 output tokens; NewtonBench additionally limits one experiment call to 20 queried items. Limits are identical across conditions within a setting.

Benchmark	Model turns	Tool calls	Tokens	Wall time
Harness-Bench	30	30	96,000	1,200 s
NewtonBench	12	10	192,000	3,600 s

B.3 Outcome and uncertainty accounting

All reported aggregates use a fixed denominator containing every declared test task. When a run yields a valid native benchmark score, we retain that score, including zero. When construction or execution fails to yield a valid result, the task contributes zero to the system-level aggregate. This rule covers missing source bindings, infrastructure failures, and unresolved technical outcomes. The Builder and Target may attempt repairs within their permitted limits, but a run that remains unsuccessful or exceeds its total budget is still counted as zero. Thus, the absence of a native score never removes a task from the reported denominator.

For paired comparisons, we resample test tasks 20K times while preserving the pairing between conditions. We report the 2.5th and 97.5th percentiles as unadjusted 95% intervals. These intervals describe variation across tasks in the fixed test inventory, rather than variability across repeated stochastic model generations.

Component ablations operate directly on frozen harnesses: we remove one component family without rebuilding the remaining support. Each paired contrast therefore measures the effect of that intervention on the recorded harness, conditional on a single official execution per task. For transfer experiments, paired comparisons use only the concurrent no-skill control from the same campaign. Differences across campaigns are reported only descriptively.

C Baseline configurations and interpretation

All conditions share the same development/test splits and per-task Target execution limits. We use GPT-5.6-Sol for both harness construction and skill extraction, with Gemini-3.6-Flash, Qwen3.8-Flash, and GPT-OSS-120B as Targets. We keep all the learned banks and notes frozen before testing.

Retrieved versus full-bank access. These variants use the same learned content and differ only in how much of it is supplied at deployment. Full-bank access provides every learned skill, whereas retrieval uses the initial public task prompt as a BM25 query ($k_1 = 1.5$, $b = 0.75$) and selects at most two positive-scoring matches. If no skill receives a positive score, the resulting pack is empty. For free-form trajectory-mined notes, retrieval operates on natural paragraphs, while full-text access provides the complete notes. Holding the learned content fixed allows this comparison to assess selective versus comprehensive delivery without changing the learning procedure.

Native environment. The Target executes directly in the shared neutral environment H_0, using native tools, default conversation history, and scratch storage. It receives no learned guidance or Builder-generated components, although it may create its own working aids through the available tools. This baseline represents ordinary task solving in the Target's native environment and provides the reference point for evaluating additional support. Gains from our method over this setting therefore reflect the combined contribution of harness construction and learned meta-skills.

No-skill Builder. To separate the contribution of construction from that of learned experience, GPT-5.6-Sol constructs a fresh harness for each public task using an empty skill pack. The Builder otherwise retains the same seven component families, construction instructions, and information about the Target as our method, together with the same allowance of 16K output tokens and at most two interface-repair attempts. This baseline preserves the Builder's ability to design support while removing access to learned meta-skills. For Gemini on Harness-Bench, performance increases from

37.35 in the native environment to 53.29 with no-skill construction, and then to 67.81 with our method. The first improvement captures the benefit of construction alone; the second captures the additional benefit of learned experience under the same construction interface.

Direct Builder skills. This baseline supplies the Target with the same meta-skill pack that the Builder receives in the corresponding variant of our method. The meta-skills' *when*, *provide*, and *use* fields are appended to the Target's instructions, and the Target is explicitly allowed to implement this guidance using its native tools and execution budget. No Builder-generated callbacks or resources are installed. The comparison thus holds access to learned knowledge fixed while changing whether a separate Builder implements that knowledge as harness support. Our full-bank method achieves higher scores in all six original settings, including 67.81 versus 42.38 for Gemini on Harness-Bench. These results support the benefit of having the Builder translate meta-skill guidance into concrete support before Target execution.

Independent Structured Target skills. This baseline learns skills specifically for the Target's own problem-solving, using a separate GPT-5.6-Sol learner with no access to our Builder's programs, trajectories, or meta-skill bank. Learning proceeds over two passes through the development set: the Target executes tasks in the neutral environment, and its current skill bank is supplied to subsequent executions as updates accumulate. Each update allows up to 8K output tokens and at most one skill addition or revision, with each skill limited to 192 tokens. The resulting bank is frozen and delivered directly to the Target at test time. Here, "independent" refers to the separately collected experience and separately learned bank; the extraction model remains the same. This baseline compares learning guidance for the Target's task-solving procedures with learning principles for the Builder's support design. Direct Target skill learning is competitive in several settings: Qwen scores 72.89 on Harness-Bench compared with our method's 74.81, while Gemini scores 67.81 on NewtonBench compared with 68.84.

Mined Free-Form Target skills. This baseline also uses GPT-5.6-Sol to extract problem-solving advice from independent Target experience, but replaces iterative structured updates with a single offline consolidation step. The Target first completes one pass through the development set, after which the learner consolidates all resulting trajectories into a note of at most 3K tokens. The learner freely chooses the content, format, and organization of this note, without prescribed skill fields or explicit add, revise, and keep operations. At test time, the frozen notes are supplied directly to the Target, with no separate harness construction or Builder-generated components. Results vary across Target–benchmark settings. On Harness-Bench, GPT-OSS scores 66.01 with free-form notes, compared with 56.63 with Independent Structured Target skills. On NewtonBench, Gemini scores 41.44 with free-form notes, compared with 67.81 with structured Target skills and 68.84 with our method. These comparisons show that the relative effectiveness of static free-form advice and iteratively learned structured skills depends on the evaluation settings and models.

D Case Studies: From Meta-Skills to Executable Support

We present two successful full-bank test episodes with GPT-5.6-Sol as **Builder** and Gemini-3.6-Flash as **Target**. They illustrate complementary forms of support design: constructing a task-specific executable checker and coordinating an existing scientific interface through memory and control. Each case traces how a learned meta-skill informs the generated harness and how that support is used during **Target** execution. These selected cases provide qualitative insight into the resulting workflows, without estimating typical success rates or isolating the causal contributions of individual components.

D.1 Making an artifact contract executable

Case 1: Builder connects artifact validation to submission

Task and support problem. A metric-migration task in Harness-Bench asks **Target** to compare old and new metric definitions, query results, and dashboard snapshots under a migration policy. It requires four deliverables: a metric comparison, a regression ledger, a summary, and a caveat report. The challenge combines interpreting changes with keeping these deliverables complete and consistent. For example, a higher activation rate may be justified by a revised denominator, whereas a retention dashboard that

still uses the old denominator violates the migration policy. Distinguishing these cases requires semantic judgment; ensuring that the resulting classifications agree across artifacts requires systematic bookkeeping.

From a learned principle to executable support. The frozen bank provides a checkpointing meta-skill that calls for tracking output updates, validation status, and remaining budget, with a prompt finish after successful validation. It leaves evidence interpretation, drafting, and repair to **Target**. **Builder** implements this principle through task-specific instructions, an executable auditor, a controller, and a submission gate. The auditor is generated during harness construction, while **Target** subsequently authors all substantive deliverables. This division assigns mechanical verification to the harness and preserves **Target**'s responsibility for the analysis.

The auditor makes consistency requirements explicit. The generated auditor checks that the metric comparison covers every metric appearing in either query export exactly once, that reported values match the sources, that the regression ledger has the required coverage, and that the summary's category counts agree with the comparison. It also verifies the presence of all required artifacts, their schemas, and numerical formatting. These checks specialize the public output requirements to the task's actual inputs. For example, once **Target** labels a metric as an unexpected regression, the auditor can check whether the ledger and summary consistently reflect that decision. Determining whether the label is justified by the migration policy remains **Target**'s responsibility.

Validation governs the submission workflow. The controller tracks writes to the four deliverables and prompts **Target** to invoke the auditor once all four have been observed. A clean audit enables submission and prompts completion unless **Target** has identified a concrete semantic defect. Subsequent edits through the tracked file-writing interface invalidate the audit result and require another check. The submission gate enforces this dependency by withholding completion until the controller records a clean audit. Together, these components connect artifact creation, validation, and submission through an explicit readiness condition.

Observed execution and outcome. In the recorded episode, **Target** reads the migration evidence, writes the four deliverables, and invokes the auditor, which reports zero errors and zero warnings. The resulting analysis covers five metrics: two expected definition changes, one unexpected regression, one with no material change, and one requiring review. The regression ledger identifies the retention dashboard's continued use of the old cohort denominator, while the missing post-migration gross-margin value is left for review. **Target** then submits successfully, achieving a native score of 1.0 with 14 model calls and 15 counted tool calls. No repair iteration is needed. On the same task, no-skill construction and Direct-all produce no output artifacts and each score 0.0435, terminating because of token-budget exhaustion and model failure, respectively.

Insight: support design separates judgment from bookkeeping. The reusable knowledge lies in recognizing which obligations can be made executable. **Builder** translates the checkpointing principle into checks for metric coverage, source-value fidelity, and agreement across deliverables. These obligations are straightforward to verify deterministically but compete for attention with interpreting migration evidence. By assigning them to generated code, the harness lets **Target** concentrate on substantive decisions while making their consistent expression across artifacts mechanically checkable.

Insight: validation becomes actionable when it controls completion. The auditor supplies a capability, while the controller and submission gate determine when to use it and how its result changes the next action. A clean check establishes a concrete stopping condition, and an observed edit reopens validation. This episode demonstrates the successful draft–audit–submit path: the constructed workflow supports delivery of a complete, consistent answer within the execution budget. The comparison illustrates the potential value of this integrated support, although a single episode does not isolate the contribution of each component.

D.2 Preserving scientific progress after computational failure

Case 2: Builder preserves evidence and bounds failed computation

Task and support problem. A NewtonBench task exposes a noise-free apparatus controlled by displacement x and mass m, which returns velocity after exponential energy loss. The requested quantity is the pre-loss elastic potential $U(x)$, so the directly calculable kinetic energy $E_k = mv^2/2$ must be corrected for the loss. Within ten tool calls, **Target** must design informative experiments, distinguish the measured observable from the requested quantity, infer the underlying law, and submit an executable function. The support problem combines a conceptual challenge—recovering latent energy from observations—with a procedural one: preserving enough evidence and budget to finish when computational analysis fails.

From learned principles to coordinated support. Three learned meta-skills guide the construction: retain raw evidence while bounding analysis, reserve budget for submission, and distinguish the requested quantity from measured observables. **Builder** implements these principles through instructions, memory, a controller, and recovery and submission checks around the benchmark's existing experimental and computational interfaces. The instructions direct **Target** to infer the energy-loss scale jointly with the displacement law and recommend duplicate inputs, cross-mass probes, and fresh measurements. The scientific hypothesis and its coefficients remain for **Target** to infer.

Memory preserves evidence independently of analysis. The memory module stores recent experiment batches together with their inputs, separately from the latest computation result. A failed fitting attempt therefore leaves the measurements available for subsequent reasoning. Retrieval supplies the latest analysis result and up to four recent experiment batches to later model requests. In the recorded run, both earlier experiment batches are delivered again immediately after the second computation failure, making the accumulated evidence available at the point of recovery.

Control bounds retries and preserves a path to submission. The controller counts failed computation calls and, after two failures, removes computation from the available tool menu while keeping experimentation and submission available. Recovery feedback directs **Target** to continue from retained measurements. When three or fewer tool or model calls remain, a finalization reminder asks **Target** to spend at most one further action on a concrete contradiction or fresh validation before preparing the submission. A separate submission check verifies the required response format and function signature. These mechanisms make recovery responsive to observed failures and remaining budget.

Observed analysis and recovery. **Target** first collects a broad experiment batch containing duplicate inputs and cross-mass probes, followed by a second batch focused on small displacements. Two successful computation calls derive kinetic energy and compare candidate loss corrections. For example, at $x = 0.01$, the observed kinetic energy is approximately 4.58571, while the candidate corrections $E_k e^{100x}$ and $E_k e^{50x}$ yield approximately 12.46524 and 7.56055, respectively. These calculations explicitly connect the measured observable to hypotheses about pre-loss energy. The next two fitting attempts fail: one uses unsupported syntax, and the other exceeds the computational instruction limit. The controller then removes computation from the tool menu, and **Target** collects ten additional measurements at displacements between 0.0001 and 0.001. With three tool calls remaining, the finalization reminder activates, and **Target** proceeds to submission.

Scientific result. **Target** submits the following law:

$$\widehat{U}(x) = \max\{0,\ 1241.9012\,x + 462.282\,x^2\}.$$

The native evaluator accepts the submission as equivalent to the reference law, reporting symbolic accuracy 1 and RMSLE 0. The run uses eight model calls and eight counted tool calls. On the same task, no-skill construction produces a valid but incorrect law, with symbolic accuracy 0 and RMSLE 0.092, while Direct-all ends with a model failure and no submission. The successful trajectory therefore includes an observed recovery from failed analysis to further measurement and completion.

Insight: recovery connects evidence retention with action control. The initial guidance to keep computation bounded does not prevent two fitting failures. The constructed support nevertheless preserves a productive continuation: memory retains the measurements, the controller recognizes repeated failure, and the available actions change accordingly. **Target** can return to experimentation without losing its earlier evidence or spending the remaining budget on further computation retries. This case illustrates how a reusable recovery principle becomes an executable policy that responds to the actual trajectory.

Insight: support components should match the bottleneck. The conceptual and procedural challenges receive different forms of support. Instructions clarify the distinction between observed and latent energy; memory maintains evidence continuity; and control limits unsuccessful analysis while reserving an opportunity to finish. Their coordination allows **Builder** to improve the use of an existing scientific interface. **Target** retains responsibility for choosing experiments and inferring the law, while the harness structures how evidence, failures, and remaining resources inform its next action.