Twin2Harness: Train Like RL. Distill into the Harness

The Rayo Team
Corresponding author: sunil@rayo.com

Abstract

Enterprise agents fail in the last mile: the policies, systems and exceptions specific to each business. Reinforcement learning fixes such failures well because it learns on-policy, from the agent's own mistakes, but stores what it learns in weights that are opaque, slow to update and tied to one model. Prompt and harness optimizers store what they learn in readable text, but learn from scalar scores or unchecked reflection. **Twin2Harness** combines the two. A frozen model explores a **Twin** of the business's systems; a **Verifier**, calibrated against grounded outcomes and acting as a critic, returns the first mistake in each run, its cause and a repair; and the repairs are distilled into the **harness**, the instructions, skills and tools around the model. We frame this as on-policy distillation into context. On 1,000 held-out real financial-operations cases, each of four models rises from 12–19% to 89–93% verified completion. On τ^3-Banking, Claude Haiku 5.5 rises from 0.265 to 0.804 pass[1], above every leaderboard entry and well ahead of SkillOpt and Exo; on AppWorld, a 27B model with frozen weights rises from 30.1% to 79.6% on the challenge split, close to the same model trained with RL (83.7%); and a harness learned on Haiku lifts GPT-6 Sol to 74% strict success on BizAgentBench. The gains hold on tasks the harness never saw, and an audit finds no test-task content in it. Learning takes under four hours and $250 per benchmark, and on two of three benchmarks a success also costs less. What is learned is plain text and code that a domain expert can read, edit and veto.

1 Introduction

Enterprises want agents that do real operational work: reviewing transaction-monitoring alerts, verifying businesses (KYB), processing supplier invoices, reconciling accounts, updating ERP records, onboarding employees. Each case runs long, spans many systems, and follows policies and SOPs that no two businesses share. Today's agents are still clumsy with the tools and blunt with the rules. On BizAgentBench, DeepSeek-V4.1-Flash makes 1.7 times as many tool calls as a reference execution, repeats identical calls 16 times per run, and opens 20 documents where 5 are needed; capped at 1.2 times the reference's calls, its strict success falls from 43% to under 1%. Agents also reach the goal by breaking the rules on the way, moving funds below a required reserve or executing a payment before it is approved: Qwen3.8-27B reaches the goal in 42% of runs but does so validly in only 33% [1]. The newest finance and enterprise agent benchmarks are far from saturated. On τ^3-Banking, whose retail-banking tasks require searching a 698-document knowledge base, the best base model on the leaderboard (Qwen 3.8 Max) completes 55% of tasks and the best custom harness (Cortex) 73% [2]; the best published agents complete 64% of long-horizon payments-operations cases (BizAgentBench) [1] and under 40% of enterprise operations tasks (EnterpriseOps-Gym) [3]. On real transaction-monitoring cases, the best frontier model completes 12% (§5.1).

These failures are a lack of knowledge, but not the kind pre-training supplies. The agent does not know what evidence this business requires before a decision, which of its rules governs the case, or

Preprint.

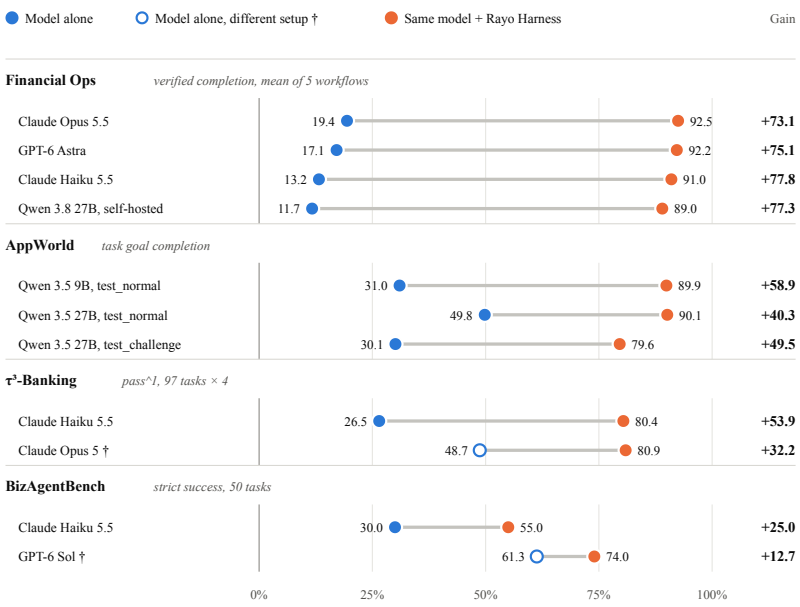

Figure 1: *The same weights, a different agent: with the Rayo Harness alone, models gain up to 78 points.* Each row is one model on one benchmark, with weights unchanged. In rows without †, both points come from our own runs under the same setup, so the harness learned in the Twin is the only difference (details in Tables 2, 3, 4 and 6). In the two rows marked †, the model-alone point comes from a different setup and is shown only for context: Claude Opus 5 alone is the leaderboard run at maximum effort, and GPT-6 Sol alone is the published result, with a harness learned on Claude Haiku 5.5.

when a case counts as done. So it clears an alert without the evidence the SOP requires, skips a required approval, asks a customer for information already on file, or escalates "to be safe" when it could have completed the case. Two separate studies of failed runs find the same pattern. On real financial-operations cases, Rayo finds wrong decisions (38%), skipped steps (24%), evidence gaps (19%), navigation failures (10%) and hallucinated details (9%) [4]. On enterprise tasks that are each solvable as written, where GPT-6.1 Sol passes 10 of 40, Collinear finds evidence gathering (39%), reasoning over evidence (30%), stopping before finishing (24%), interface errors (5%) and arithmetic slips (2%), with the same groups leading across 20 models [5]. In both, most failures (81% and 93%) are about what to gather, how to decide and when a case is done; only 5% to 10% come from operating the interface. That is knowledge of the business's procedure, and it is what a harness can hold. This last mile does not shrink as models improve. Every business has its own policies, systems, exceptions and definition of done, little of it appears in any pre-training corpus, and it changes as products, regulations and customers change. It has to be learned per business, and learned again whenever the business changes.

The harness. By *harness* we mean everything in an agent that is not the model. Following Krentsel et al. [6], it is the scaffolding around the model that assembles the model's input and executes and observes its actions: the instructions and prompts, skills, tools, context and memory management, and the runtime that ties them together. The model's weights hold general ability; the harness is where a business's procedure can live.

Two ways to learn the last mile, each half right. *Reinforcement learning learns well.* The agent acts, and it is corrected in the states it actually reaches, so it fixes the mistakes it really makes; on AppWorld, RL-trained 14B and 27B models lead the leaderboard [7; 8]. For many enterprise workflows, though, training weights is overkill. What the agent lacks is a business's procedure, and weights are a costly place to keep it. Procedure keeps changing: SOPs are revised, audit rules evolve, and new scenarios emerge. A domain expert team needs to update the agent when that happens, and to see what it was taught, but weights take GPU-days to retrain and cannot be read or edited by the people who own the procedure. Base models also now arrive faster than a business can retrain

them. In the last twelve months Qwen shipped five model generations (3.5, 3.6, 3.7, 3.8 and 4) and GLM five (4.7, 5, 5.1, 5.2 and 5.3), roughly one every two to three months [9; 10], and each new base model means training again. Weights are the right place for general knowledge that carries across businesses and workflows: using tools reliably, reading documents, reasoning over evidence, following instructions. That knowledge is stable, it benefits every customer, and it is exactly what each new base model improves. A business's procedure belongs in a layer above it, where it can change as fast as the business does. The two layers combine well (§7). *Prompt, skill and harness optimizers store well.* They improve a frozen model through its prompts, context, skills or code [11; 12; 13; 14; 15; 16], and what they learn is text or code that can be read and moved to another model. The harness matters as much as the model: on AgenticFinBench, harness design explains more of the variation in evidence grounding than model choice does [17]. Reflective optimizers such as GEPA already turn the agent's own trajectories into natural-language diagnoses [11]. What is missing is a check on those diagnoses: the reflection is a model's unverified reading of what went wrong, and the score it is anchored to comes from a benchmark grader that a real business does not have.

Train like RL, distill into the harness. We take the learning signal of one and the storage of the other. Think of how a business trains a new analyst: the analyst works real cases, a senior reviewer corrects the mistakes the analyst actually makes, and the lessons go into the team's playbook rather than staying in one person's head. Harness training does the same for an agent. The agent works cases in a copy of the business's systems and makes its own mistakes, as in RL. A critic calibrated on ground truth, the Verifier, finds the first mistake in each case, as in actor-critic RL, and says why it happened and what the right step was, as in on-policy distillation [18; 19; 20]. The corrections are written into the harness, not the weights. A new version takes hours, not GPU-days. A revised SOP is an edit, not a retraining run. A new base model inherits everything the last one learned. The agent trains like RL; the business owns what it learned. We call this *on-policy distillation into context*, and the method Twin2Harness (§2).

What makes it possible. On-policy learning needs two things enterprises lack: somewhere the agent can act and fail without consequence, and a teacher that can judge the states the agent actually reaches without an answer key. We build both (Figure 2). A **Twin** is a resettable copy of the business's systems. A **Verifier** judges each attempt against the business's own authorities and returns the first mistake, its cause and a repair. Neither can be trusted alone, so we calibrate them against each other over a few iterations: the Twin replays the Verifier's repairs and supplies mutated runs it must reject, and the Verifier's disagreements expose faults in the Twin and its tasks (§3). In deployment the teacher draws on recorded expert work with human ground truth; on public benchmarks, on other agents' passing runs or the benchmark's own documents (§2.3).

Contributions.

1. **Twin2Harness: harness training as on-policy distillation into context**: a way to turn an agent's own failures into auditable procedural knowledge. A Verifier calibrated against grounded outcomes identifies each actionable failure and its fix, and the fixes are stored as rules and tools an expert can audit, with frozen weights (§2).

2. **The Twin and the Verifier that make it possible**, with a calibration protocol that checks each against the other and measures the Verifier on the issues it names and on whether its repairs succeed when replayed (§3).

3. **Evidence on real cases and three public benchmarks** (§5). On 1,000 held-out real financial-operations cases, each of four models rises from 12–19% to 89–93% verified completion averaged over five workflows. On public benchmarks chosen to share their challenges, a 27B model with frozen weights is competitive with the same model fully RL-trained on AppWorld, and a harness learned on one model transfers to another. The gains hold on cases the harness never saw, so it learns procedure rather than memorizing answers. Every result is labelled with its protocol and exposure (Table 9).

2 Learning: on-policy distillation into the harness

2.1 Setup

A business can rarely provide a full dataset of correct executions. What it has is sparse evidence about the work: trusted input and output examples, reference trajectories where they exist, SOPs, explicit business rules, and the expected form of the output. A trusted input and output pair fixes an endpoint, not every valid way to reach it, which is why the agent must be taught a procedure and not just shown answers.

A frozen model M works a case t in an environment E through a harness H, everything in the agent other than the model's weights. Each run produces an execution τ, the sequence of actions and results:

$$\tau \sim p_{M,H}(\cdot \mid t, E). \tag{1}$$

Here $p_{M,H}$ is the distribution over executions that running M with harness H induces; it is random because the model samples its actions. The Verifier V, which plays the role of a critic, reads the execution and returns a verdict

$$V(\tau) = (y,\, d,\, c,\, r), \tag{2}$$

where $y \in \{0, 1\}$ is the outcome, d the step where the run first went wrong, c the cause, and r the right step there. The weights of M, the environment and the grader are fixed; only H changes. The goal is to raise success on cases the harness was not built from, with the weights frozen.

2.2 Train like RL

Harness training shares what makes RL work for long-horizon agents: it learns from the agent's own runs, at the states the agent actually reaches, and the feedback on those runs changes what the agent does next.

Learning in the agent's own states. An agent trained to imitate expert traces only ever sees the states an expert visits. At deployment its first small error takes it somewhere the expert never went, where it has no guidance, and errors compound over a long case; learning at the states the learner itself reaches is the standard remedy [18; 21]. Enterprise cases are long: a single one can run for hours, move across many systems and take hundreds of tool calls, so an agent corrected only on expert states drifts. Every correction in harness training is made at a state the agent actually reached.

A richer signal per mistake. RL and on-policy distillation both learn from the agent's own runs, but get very different feedback from each one (Appendix G gives both updates). RL gets a reward, at most one bit per case, and must work out for itself which of a hundred steps caused a failure, which is why it needs many rollouts and GPU-days. On-policy distillation gets a teacher's choice at every state. The Verifier goes one step further for a harness: it names the step d where the run first went wrong, the cause c and the right step r, so the credit assignment RL must learn is done by the teacher. A few rounds of runs are then enough to make progress, which is why a version takes hours. Unlike both, harness training changes the harness, not the weights.

Table 1 puts the two halves of the argument side by side: the left half is why harness training trains like RL, the right half is what distilling into the harness buys. §5.2 tests the first against RL directly.

2.3 Distill into the harness

Harness training keeps the on-policy sampling of RL and on-policy distillation but replaces the gradient step with a change to the harness. With the model frozen, the harness is the agent's trainable parameter, a second set of weights written in language and code, and its optimizer is itself an LLM. In round k, the Verifier's diagnoses of failed executions are consolidated into one change Δ_k, so that it addresses recurring patterns rather than single cases:

$$\Delta_k = \mathrm{Consolidate}\Big(\big\{\, (d, c, r) = V(\tau) \;:\; \tau \sim p_{M,H_k},\; y(\tau) = 0 \,\big\}\Big). \tag{3}$$

Table 1: A conceptual comparison of how each approach learns and where it stores what it learns. Harness training combines what the other rows offer separately: learning from the agent's own runs, a dense diagnosis calibrated against grounded outcomes, and storage that experts can read and the next model can use.

	How it learns		*Where it stores*		
Method	Learns from	Signal per run	Updates	Expert-readable	Survives a new model
SFT on expert traces	expert's runs	every expert step	weights	×	×
RL	own runs	one reward (≈1 bit)	weights	×	×
On-policy distillation	own runs	teacher's action, every step	weights	×	×
Prompt / skill optimizers	often own runs	a score, success label or unchecked reflection	prompts, skills	✓	mostly
Twin2Harness (ours)	**own runs**	**calibrated diagnosis: first mistake, cause, fix**	**harness**	✓	✓

Here H_k is the harness in round k, the set collects the Verifier's verdicts on the executions that failed ($y = 0$), and $\mathrm{Consolidate}$ merges diagnoses that share a cause. The repair r draws on a reference where one exists (experts' recorded work in deployment; on the benchmarks, other agents' passing runs or the benchmark's own documents), but the agent never imitates it: the reference only shows the right step where the agent went wrong.

Who proposes, what can change, how it is tested. A builder model (Claude Opus 5.5 in our experiments) reads the consolidated diagnoses and the current harness, and proposes the smallest change that addresses each cause. A change can add guidance the agent reads (a rule tied to the policy it comes from, a check before a consequential action, a missing step), a tool the agent can call, or a change to the runtime around the model; it never touches the model, the Twin, the Verifier or the cases. The candidate harness then reruns, in the Twin, the same cases as the current one, both those that failed and those that passed, from the same starting states, and is kept only if it does better, judged in a strict order:

$$H_{k+1} = \begin{cases} H_k \oplus \Delta_k & \text{if } H_k \oplus \Delta_k \succ H_k, \\ H_k & \text{otherwise,} \end{cases} \tag{4}$$

where $H_k \oplus \Delta_k$ is the harness with the change applied, and $\succ$ ("does better than") compares two harnesses on the same cases, criterion by criterion in this order, moving to the next only on a tie:

$$\text{constraints} \succ \text{grounded outcome} \succ \text{no regression} \succ \text{judgment} \succ \text{process} \succ \text{cost}.$$

A change must first break no hard constraint the business imposes, then lose no case that ground truth says the agent got right, and only then do the Verifier's judgment, the quality of the process and the cost count. Efficiency cannot buy correctness, and the Verifier cannot overrule a grounded failure. Equation (3) plays the role of a teacher's labels; equation (4) is the acceptance step that stands in for a gradient update.

A worked edit (illustrative, paraphrased from an AppWorld failure).

Observed failure. The agent updates a record through an app's API. The call returns an error, but the agent goes on to the next step, which depends on the update, and reports the task done.

Diagnosis. First mistake: the step after the failed call. Cause: the agent treats a write as done without reading its response, so later decisions assume a state change that never happened. Repair: confirm a write succeeded before any step that depends on it.

Harness change. One rule added to the guidance, and one helper added to the agent's tools:

```
+ After any call that changes state, confirm it succeeded: an explicit success in
+ the response and, where it matters, the new state read back. If it failed, fix
+ it before any step that depends on it. If the outcome is unknown, read the state
+ before retrying, so a write is never applied twice. Never report a change that
+ did not happen.

+ def confirm_write(response, read_back=None):
+     """Stop unless a state-changing call clearly succeeded."""
+     ok = (isinstance(response, dict) and response.get("success") is True
+           and "error" not in response)
+     if not ok:                          # failed or ambiguous: never assume success
+         raise WriteNotConfirmed(response)
+     if read_back is not None and not read_back():
+         raise WriteNotConfirmed("state did not change")
+     return response
```

Acceptance. The edit is accepted only if replay in the Twin fixes the targeted failures without regressing previously successful tasks.

This is the reverse of context distillation [22], which moves what a prompt teaches into the weights. Keeping the corrections in the harness is what makes them fast to update, readable by the experts who own the procedure, and portable to the next model.

3 What makes it possible: the Twin and the Verifier

Learning on-policy means the agent must explore and make its own mistakes, so it needs somewhere to make them safely and a teacher that can judge states no reference ever visited. The Twin and the Verifier provide the two.

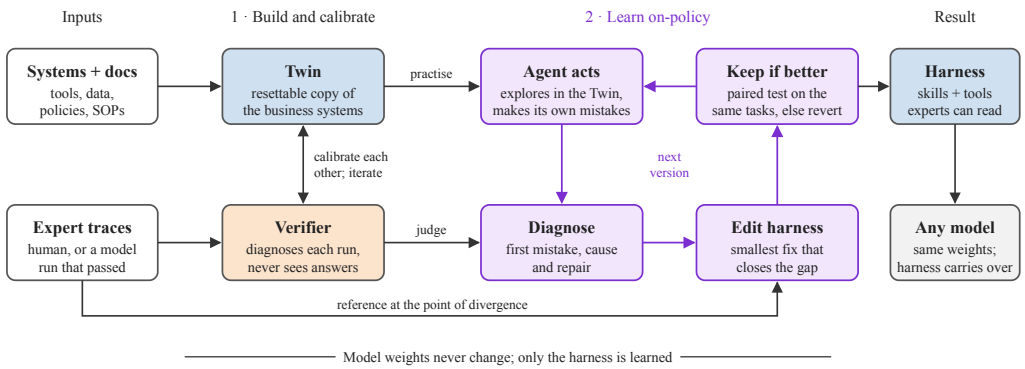

Figure 2: *The Twin2Harness training process.* Rayo builds a Twin and a Verifier from a business's systems, documents and expert traces, and calibrates them against each other (stage 1). The agent then learns on-policy (stage 2): it acts in the Twin, the Verifier diagnoses its first mistake, and a harness edit is kept only if it does better on the same tasks.

The process has two stages. **Stage 1 builds and calibrates the Twin and the Verifier.** Both need calibration, and each is calibrated partly against the other; fixing one often exposes the next flaw in the other, so stage 1 takes a few iterations and ends only when both pass their checks. **Stage 2 learns on-policy** (§2). In actor-critic terms (§3.2), stage 1 trains the critic and stage 2 trains the actor against it. Ground truth is used to calibrate the Verifier in stage 1, taken from labeled training cases where they exist and created in the Twin where they do not (§3.2); in stage 2 the Verifier sees only what the agent saw, and its edge is the calibration it carries and the hindsight of reading the whole run.

Learning runs on two timescales. In the *inner loop*, only the harness changes: the model, the Twin, the Verifier and the cases are frozen, so every candidate change is compared on the same footing and the Verifier cannot drift toward the agent. In the *outer loop*, the improved agent's own runs surface

near-misses no synthetic case anticipated, much as dataset aggregation collects the states a learner actually reaches [18]; labeled by ground truth, never by the current Verifier alone, they improve the Twin, the Verifier and the cases as new frozen versions. A gap in the Twin or the Verifier is never fixed by editing the harness.

3.1 The Twin

A **Twin** is a private, resettable replica of the part of a business where the work happens: the applications people use, the data and state behind them, the policies that govern them, and, where the work involves a customer, a simulated counterpart. The agent works in it through the same interfaces a person would, and each action changes the Twin's state as it would change the real systems. Three properties make it a Twin rather than a mock:

- **Faithful.** It returns the same results and the same grades as the real systems.
- **Resettable.** Any case can be run again from the same starting state, so runs can be compared.
- **Private.** Mistakes made while learning never reach a customer.

Built from the work itself. In deployment, a Twin is built from what a business already has: recordings of experts doing the work, the written SOPs, and the systems themselves [4]. The same recordings supply the cases the agent practises on, the expert outcomes that ground the Verifier, and a first harness drawn from how experts actually work. On the public benchmarks, the Twin is a copy of the benchmark's own environment.

Calibrated by replay. A Twin is only useful if it is faithful, so known work is replayed until the Twin reproduces the same results and grades. Where cases are few or miss important situations, the Twin also hosts synthetic cases, each admitted only after checks that it is solvable, consistent and does not give away the answer.

3.2 The Verifier: a calibrated critic

The Verifier reads a finished run and judges it against the business's own authorities: its policies, procedures and system rules. It never judges against the agent's harness and never sees the answer. For each run it returns an outcome, the first mistake, its cause and a repair.

A critic. The closest analogy is actor-critic RL [23; 24]. The harness is the actor and the Verifier the critic: it says which of the actor's actions made the outcome worse, as an advantage estimate does in A2C. Two things differ. The critic returns a diagnosis rather than a number, and the actor is updated by an edit rather than a gradient. Because the critic names the better action, its judgment can be distilled into the harness (§2.3).

Calibrated before use. A critic is only as good as the outcomes it was trained against, so the Verifier is calibrated on cases whose outcome is known. It is held to three standards: agreement on the outcome, agreement on what went wrong, and repairs that actually work. The last two matter most, because a critic that is right about the outcome for the wrong reason points the harness at the wrong fix. Calibration is the only point at which the Verifier sees more than the agent. During learning it sees what the agent saw, and its edge is what calibration taught it and the hindsight of reading the whole run. That is why the method keeps working where no grader exists.

Where the ground truth comes from. From existing labels, a benchmark's checks or experts' recorded outcomes where they exist; elsewhere, from cases built in the Twin whose correct outcome is known in advance. These include both good and bad work, so the Verifier is tested on both.

Kept honest. Ground truth outranks the Verifier: where a deterministic check or a grounded outcome exists, it decides, and the Verifier never labels the data used to recalibrate itself. And because a fixed critic can be gamed, as over-optimized reward models are [25], we measure how often it accepts a run that ground truth G fails, alongside how often it rejects good work:

$$\mathrm{FA}(X) = \Pr_{\tau \in X} \big[\, V(\tau) = \text{accept} \ \wedge \ G(\tau) = \text{fail} \,\big], \tag{5}$$

where X is a set of runs with known ground truth, $V(\tau) =$ accept means the Verifier's outcome y in (2) is 1, $G(\tau)$ is the grounded result, and the false-rejection rate is defined the same way with the verdicts swapped.

3.3 Calibrating each against the other

Neither component can be trusted alone, and each is the other's test. The Twin replays the Verifier's repairs and supplies deliberately broken runs that the Verifier must reject; the Verifier's disagreements expose faults in the Twin, its grading or its cases. Fixing one often exposes the next flaw in the other, which is why stage 1 iterates until both pass their checks (§5.2 and §5.3 report what this caught).

4 Experimental setup

We evaluate in two settings. The **Financial Operations Benchmark** is as real as it gets: real cases from real businesses, with human ground truth and expert grading (§5.1). Its data cannot be released, so we also evaluate on three **public benchmarks** that share its core challenges and have well-known prior results: AppWorld, where an agent works across many applications autonomously and end to end; τ^3-Banking, where it serves customers under policies spread across a large knowledge base; and BizAgentBench, which tests what enterprises need most, following and enforcing policy at every step. Appendix A summarizes the benchmarks and how each result was produced.

AppWorld. AppWorld asks an agent to complete everyday tasks across 9 apps and 457 APIs by writing code, with no user to ask; success is checked by the benchmark's unit tests on the final state of the apps. We learn from the train (90) and dev (57) splits and evaluate on test_normal (168) and test_challenge (417). To cover more of the apps and more of the situations an agent may meet than the training splits contain, we also generate synthetic tasks in the Twin. During the test runs, the harness was allowed to adapt using its own Verifier's diagnoses of the agent's runs, as it would in deployment, and as Exo does by design [6]. It was never allowed to see the benchmark's ground truth. We study Qwen 3.5 9B, Qwen 3.5 27B and Claude Haiku 5.5, each starting from the official ReAct prompt.

τ^3-Banking. τ^3-Banking asks an agent to serve banking customers under policies spread across a 698-document knowledge base, using tools it must discover; a simulated customer (GPT-5.2, the benchmark default) holds information the agent must ask for. The benchmark has no official split, so we split its 97 tasks 60/40 into training and held-out tasks, with no held-out task sharing a customer or record with a training task. We start from the public leaderboard trajectories of other agents on the training tasks. From these we generate synthetic tasks, calibrate the Verifier, and then learn the harness on-policy, from the target model's own runs. The held-out tasks never inform a change. We study Claude Opus 5 and Claude Haiku 5.5, each starting from τ's plain agent.

BizAgentBench. BizAgentBench asks an agent to complete long-horizon payments-operations requests for a simulated customer while following the company's policies at every step; success is strict, requiring the goal, a valid process and the protection constraints together. We read 10 of its 50 tasks to understand the benchmark. From its policies and tools we then generated 150 synthetic tasks in the Twin, and calibrated the Verifier and learned the harness on those. Synthetic tasks never count toward a score; reported results are on all 50 of the benchmark's own tasks. The simulated customer is Claude Opus 5.5, which differs from the published setup, so published numbers are context rather than a direct comparison. We study Claude Haiku 5.5, and transfer the harness learned on Haiku to GPT-6 Sol without further learning; both start from the benchmark's default prompt.

5 Results

Figure 1 summarizes the results: on every benchmark, the same model does far better with the Rayo Harness. The sections below give each benchmark in detail.

5.1 Real cases: the Financial Operations Benchmark

The Financial Operations Benchmark [4] tests agents on 1,000 held-out real cases across five workflows: transaction monitoring, onboarding, automated QC, banking operations and payment operations. Each case is run three times and each run is scored by a domain expert; a run counts as complete only if the outcome is right, every SOP step is met, and no human handoff was needed. Both arms start from the same SOP and the same systems; only harness training differs. The harness is learned from experts' recorded training cases and from synthetic cases built in the Twin; the 1,000 test cases are never used. The data are private, so we report aggregates.

Table 2: Financial Operations Benchmark: verified completion (pass@1, %) on held-out real cases, out of the box → with the Rayo Harness, per workflow and model. The last row averages each model over the five workflows.

Workflow	Claude Opus 5.5	GPT-6 Astra	Claude Haiku 5.5	Qwen 3.8 27B
Transaction monitoring	12.0 → 85.6	11.5 → 84.2	5.0 → 80.0	3.0 → 76.6
Onboarding (KYB)	18.0 → 89.1	12.1 → 89.9	14.0 → 87.1	11.2 → 84.3
Automated QC	10.5 → 98.1	10.4 → 98.0	4.2 → 97.6	4.5 → 97.6
Banking operations	26.4 → 92.8	23.5 → 92.0	18.0 → 93.1	14.0 → 90.2
Payment operations	30.0 → 97.0	28.0 → 97.0	25.0 → 97.1	26.0 → 96.2
Mean over workflows	19.4 → 92.5	17.1 → 92.2	13.2 → 91.0	11.7 → 89.0

Out of the box, no model completes more than a fifth of cases (12% to 19%). After training in a Twin, every model completes 89% to 93%, with nearly the same gain on each (73 to 78 points), and a self-hosted 27B model completes more than four times as many cases as the best frontier model did out of the box. Transaction monitoring is the hardest workflow: the best model completes 12% of cases out of the box and 86% after training.

5.2 AppWorld

AppWorld tests whether harness training can stand in for RL: whether a model whose weights never change can reach the performance of models fully trained with RL.

Table 3: AppWorld, frozen weights. TGC = task goal completion; SGC = scenario goal completion; percent, mean ± standard deviation across 3 trials per task. Qwen 3.5 9B alone and Claude Haiku 5.5 + Rayo Harness ran 1 trial on test_normal only. Published RL results report no error bars.

System	test_normal TGC	test_normal SGC	test_challenge TGC	test_challenge SGC
Qwen 3.5 9B	31.0	21.4	n/a	n/a
Qwen 3.5 27B	49.8	32.4	30.1	17.2
Qwen 3.5 9B + Rayo Harness	89.9 ± 0.3	80.4 ± 0.2	74.1 ± 1.1	59.0 ± 0.8
Qwen 3.5 27B + Rayo Harness	**90.1 ± 0.3**	**82.7 ± 2.0**	**79.6 ± 4.6**	**64.5 ± 2.9**
Claude Haiku 5.5 + Rayo Harness	95.8	94.6	n/a	n/a
CANOPY, Qwen3 14B, RL fine-tuned	86.9	80.4	67.6	50.4
SynthAPI, Qwen 3.5 27B, RL on 458K environments	88.7	83.9	83.7	71.2

On the same base model, the harness is competitive with RL. Qwen 3.5 27B + Rayo Harness uses the same base model as SynthAPI, which was fully RL-trained on 458K synthesized environments. On its own, the 27B model completes 49.8% of test_normal and 30.1% of test_challenge tasks. With the harness and no weight update at all, it rises to 90.1% and 79.6%, and comes close to the RL-trained model: 90.1% vs 88.7% TGC on test_normal, and on test_challenge 79.6% ± 4.6 vs 83.7% TGC and 64.5% vs 71.2% SGC. **Even a much smaller model gets very close.** Qwen 3.5 9B + Rayo Harness scores 89.9% TGC on test_normal, above both RL-trained models, and on test_challenge it exceeds CANOPY's RL-trained 14B model (74.1% vs 67.6%), with a third of the parameters of the 27B model and none of the RL compute. With a frontier model, the same approach goes further: Claude Haiku 5.5 + Rayo Harness completes 95.8% of test_normal tasks and 94.6% of its scenarios, the highest on that split in Table 3.

The fixes came from the agent's own mistakes. Later harness versions learned only from the Verifier's diagnoses of the model's own runs, and the fixes that mattered most were for mistakes no reference trajectory contains, because an expert never makes them: taking an amount from the

wrong field, reading a reply too narrowly, carrying on after a failed write. Only an agent that is allowed to act, and to fail, reveals them.

Repeated mistakes such as arithmetic and date handling were fixed more reliably by a tested tool than by a written rule. And once calibrated, the Verifier's repairs worked when replayed in 12 of 13 cases, which is what let the harness learn from its diagnoses rather than from ground truth.

5.3 τ^3-Banking

τ^3-Banking tests whether what is learned carries over to tasks the harness was not built from.

Table 4: τ^3-Banking, final harness, all 97 tasks $\times$ 4 trials, graded by tau v1.0.1 as on the leaderboard. The Haiku rows are the run submitted to the leaderboard. The harness was learned on Claude Haiku 5.5 and runs unchanged on Claude Opus 5.

System	pass^1, all 97	pass^3, all 97	pass^4, all 97
Claude Haiku 5.5	0.265	0.144	0.124
Claude Haiku 5.5 + Rayo Harness	**0.804**	**0.670**	**0.619**
Claude Opus 5 (low effort) + Rayo Harness	0.809	0.711	0.691
Claude Opus 5, leaderboard (max reasoning)	0.487	0.345	0.320
Cortex 0.1, regraded (1 trial)	0.732		
Qwen 3.8 Max, regraded (4 trials)	0.552	0.397	

With the Rayo Harness, Claude Haiku 5.5 rises from 0.265 to 0.804 pass^1, above every leaderboard entry in the table, including Claude Opus 5 at maximum reasoning (0.487). It also becomes far more reliable: pass^3 rises from 0.144 to 0.670, well above the best leaderboard pass^3 (0.397). On the same run, the harness scores the same on the held-out tasks (0.806) as on the training tasks it was built from (0.803), which suggests it is not overfitting them. We have submitted these results to the official τ^3-Banking leaderboard as a pull request (sierra-research/tau2-bench#759) that includes all of our trajectories, so they can be inspected and verified independently. The study also showed why the Verifier must be calibrated against the domain rather than the agent it judges: while its checks came from the agent's own skill, it let many failed runs through, and it caught them only once those checks were rebuilt from the bank's own policies. A critic that shares the actor's blind spots approves the actor's mistakes.

Against other ways of learning a harness. We compared the Rayo Harness with the two closest methods on the same model and the same training and held-out tasks (Table 5). SkillOpt learns skill edits from task rewards and the optimizer's reflection; Exo lets the agent edit its own harness at runtime, reflecting on grader feedback. All three keep the weights frozen. We did not run GEPA [11] separately: SkillOpt reports beating it in all 42 settings it compares, six benchmarks across seven models [13], so SkillOpt stands in for reflective optimizers anchored on a task score. SkillOpt lifts held-out pass^1 from 27.8% to 44.4%, Exo leaves it at or below the plain agent, and the Rayo Harness reaches 80.6%. The gap is not a stronger teacher: SkillOpt's optimizer and our builder are the same model (Claude Opus 5.5). Nor were the held-out tasks out of reach: three-quarters have a close counterpart among the training tasks, and on those Exo passes 19% and the Rayo Harness 83%, although Exo's feedback after every failed training task listed the expected actions. This compares systems, each used as designed, not learning signals in isolation; they also differ in what they can see and change and in setup (Appendix B).

It generalizes, and it is not luck. The Rayo Harness does better than every other technique and every model we compare against. On all 97 tasks, Claude Haiku 5.5 with the harness scores 0.804 pass^1, above every entry on the public leaderboard, including the strongest custom harness (Cortex, 0.732) and the strongest model (Claude Opus 5 at maximum reasoning, 0.487). The gain holds under repetition: the agent solves 62% of tasks in all four trials (pass^4 0.619), against 0.320 for Opus 5, and the best pass^3 on the leaderboard is 0.397. It also holds on tasks the harness never saw: on the same run, the held-out tasks score 0.806 and the training tasks 0.803, and on the same held-out split it nearly doubles the best other harness-learning method, SkillOpt (80.6% against 44.4%, Table 5). The reason is what the harness learns. It does not memorize tasks: an audit of the final harness finds no task identifiers, customers or case-specific values (Appendix C). It learns the domain's operating procedures (which policy governs a request, what must be checked before acting, when a case is

Table 5: τ^3-Banking, Claude Haiku 5.5: an exploratory comparison of harness-learning systems, trained on the same training tasks and evaluated on the same held-out tasks (60/40 split); pass[1]. Each system is run as it is designed to be used, so they differ in what they can see and change and in setup, not only in learning signal (see text). The plain and Rayo rows are the held-out subset of the Table 4 runs; the SkillOpt and Exo rows are separate runs.

Method	Learning signal	pass[1], held out
Claude Haiku 5.5	none	27.8
Claude Haiku 5.5 + Exo [6]	the agent's own judgment, editing its harness at runtime	19.4
Claude Haiku 5.5 + Exo with skills [6]	the agent's own judgment, plus a memory of past runs	27.8
Claude Haiku 5.5 + SkillOpt [13]	task reward and reflection	44.4
Claude Haiku 5.5 + Rayo Harness	**Verifier + on-policy harness training**	**80.6**

done) and builds tools for the steps the model keeps getting wrong, such as calculations and keeping track of the case. Those carry to new tasks and to a new model: run unchanged on Claude Opus 5, the same harness reaches 0.809 pass[1] and 0.691 pass[4].

5.4 BizAgentBench

BizAgentBench tests whether an agent follows policy at every step, and whether a harness learned on one model carries over to another.

Table 6: BizAgentBench. Goal, process and protection are scored separately; strict requires all three. Percent, on all 50 tasks.

Model	Goal	Process	Protection	Strict
GPT-6 Sol + Rayo Harness	**74.0**	96.0	98.0	**74.0**
Claude Haiku 5.5 + Rayo Harness	60.0	95.0	100.0	55.0
Claude Haiku 5.5	30.0	90.0	100.0	30.0
Claude Opus 5 [1]	64.0	96.0	98.0	64.0
GPT-6 Sol [1]	64.7	94.7	100.0	61.3

More work done, without cutting corners. On Haiku, the harness raises goal success from 30% to 60% and strict success from 30% to 55%, while process rises from 90% to 95% and protection stays at 100%. The agent completes twice as many cases and follows policy more closely while doing it. This is what correctness-first acceptance (4) is for: a change that completes more cases by breaking a rule is never kept.

Learned from the documents, not the tasks. We read only 10 of the 50 tasks to understand the benchmark. Everything else the harness learned came from 150 synthetic cases built in the Twin from the benchmark's policies and tools, and the result held on the benchmark's own tasks. A business does not need a large labelled set to start; its documents and systems are enough to build the practice ground.

Learn on a small model, deploy on any. The harness was learned on Claude Haiku 5.5 and then given, unchanged, to GPT-6 Sol. With it, GPT-6 Sol reaches 74% goal and strict success, against 61.3% to 64.0% strict for the best published configurations, with no learning on GPT-6 Sol itself, and with no policy violation on any case it completed. The expensive part, learning, runs on the cheap model; the result serves whichever model a business chooses. GPT-6 Sol's model-alone result is the published one, under the benchmark's own setup and customer simulator, so the gain is indicative.

5.5 Efficiency

A harness should make an agent cheaper to run, not just more accurate. Most of what an agent without one spends goes into failure: searching for the right tool or document, retrying an approach

that does not work, wandering until it runs out of steps. A harness learned from the agent's own mistakes removes those detours. Table 7 measures the effect per successful task.

Table 7: Effort per successful task with the Rayo Harness, relative to the model alone (ratio; below 1 means less). Tokens are input plus output; input is over 95% of the total. AppWorld: Qwen 3.5 9B on test_normal (168 tasks); a model call is one agent turn; cost is priced at Qwen 3.5 27B rates. τ^3-Banking: Claude Haiku 5.5 on all 97 tasks $\times$ 4 trials. BizAgentBench: Claude Haiku 5.5 on all 50 tasks, strict success. The customer simulator is the same in both arms of each benchmark.

		Per success, Rayo agent ÷ model alone		
Benchmark	Success, model alone → Rayo agent	Model calls	Tokens	Agent cost
AppWorld	31.0% → 89.9%	×0.18	×0.52	×0.54
τ^3-Banking	26.5% → 80.4%	×0.44	×0.42	×0.36
BizAgentBench	30% → 55%	n/a	×1.7	×1.2

On AppWorld a success takes about a fifth of the model calls and half the tokens; on τ^3-Banking, less than half of both and about a third of the cost. The guidance is not free: the agent reads it on every call, so on a task both arms solve the harness uses more input. The saving comes from removing the wasted work around it: each run is more direct, and far fewer runs are spent on failures, the longest runs of all. BizAgentBench is the exception: its guidance is long and reread on every turn of a long conversation, so a success takes 1.7× the input tokens, though prompt caching holds the cost increase to 1.2×. Learning is a one-time cost of under four hours and at most $250 per benchmark. Appendix D gives dollar costs and break-even.

6 Related work

On-policy learning. DAgger trains on expert labels at the states the learner itself visits [18]; GKD brings this to language models [19], and on-policy distillation has been reported to match RL on reasoning at a fraction of its compute [20]. RL on agent environments now leads benchmarks such as AppWorld [7; 8]. These methods learn on-policy but store in weights, as does context distillation, which moves what a prompt teaches into the weights [22]. We run the on-policy loop toward the opposite target, with a critic [23; 24] calibrated on ground truth instead of learned from returns.

Prompt, harness and skill optimization. GEPA evolves prompts by reflecting on trajectories [11], ACE grows a context playbook [12], Meta-Harness searches over harness code [16], and Exo lets an agent edit its own harness at runtime [6]. SkillOpt [13], Trace2Skill [14] and SkillGen [15] learn natural-language skills for a frozen model from its own trajectories, and other methods learn executable skills [26; 27; 28]. Our edit-and-gate step is close to SkillOpt's held-out gate and SkillGen's intervention test. What we add is the source of the signal: a Verifier, calibrated against a Twin, that diagnoses the first mistake at the agent's own states, so learning works where no benchmark grader exists. Appendix E covers further work on weight training, harness and skill optimization, environments and benchmarks, and LLM judges.

7 Discussion and limitations

The teacher is whoever completed the case. Part of what the harness holds is the judgment behind runs that completed cases the student failed: in deployment, human experts' recorded work; on the public benchmarks, other agents' passing runs or the benchmark's own documents, written down for the student. We see this as the point: it is how expert judgment reaches an agent without changing its weights. But the teacher's cost belongs in the learning budget, and whether the student can teach itself is open.

What the harness cannot fix. It fixes procedure, planning and bookkeeping. It cannot fix a context window the harness does not fit, a simulator that ends the conversation, or a grader quirk; those are logged against their component, not fitted.

Harness and weights are complementary. Harness training comes first, and it produces what weight training needs: a Twin, admitted tasks, diagnosed executions and a calibrated Verifier.

Where it fits. The method suits work whose outcomes can mostly be checked automatically, as most operational work in finance and enterprise systems can; workflows that need an expert to label every outcome are a poorer fit (Appendix F).

Twins built from the outside. Each Twin on the public benchmarks is a copy of the benchmark's own environment, whose code we could read, so its fidelity was easy to get right. A business's systems are not like that: they are black boxes, and in deployment the Twin has to be built from the outside, from recordings of expert work and the systems' own interfaces. The real-case results come from Twins built this way, but there the fidelity of the Twin cannot be measured against a published environment. Building a Twin from the outside for a public benchmark, and measuring how faithfully it reproduces the real results and grades, is the next test. We aim to run it on more enterprise benchmarks, starting with EnterpriseOps-Gym [3] treated as a black box, then CRMArena-Pro [29], AgenticFinBench [17] and FrontierCUA [5].

8 Conclusion

The last mile of enterprise work does not need new weights. It needs a place to practise and a teacher that can tell the agent where it went wrong. Twin2Harness provides both, and the results are not close. On 1,000 real financial-operations cases, every model we tested, from frontier models to a self-hosted 27B, went from completing at most a fifth of cases to completing about nine in ten. On τ^3-Banking, a small model with a learned harness beats every entry on the public leaderboard, including the strongest frontier model, solves 62% of tasks in all four trials, and scores the same on tasks it never saw. On AppWorld, a 27B model with frozen weights rivals the same model trained with RL on 458K environments, and a 9B model completes more test_normal tasks than either RL-trained model.

This moves where learning happens. RL's two strengths, learning on-policy and learning from one's own mistakes, survive when the gradient step is replaced by an edit to the harness, and a calibrated critic does the credit assignment that RL leaves to many rollouts. Learning takes hours, not GPU-days, and a few hundred dollars per benchmark. The open questions move from the optimizer to the environment and the critic: how faithful a Twin built from the outside can be, and whether an agent can become its own teacher.

It also changes who owns the agent. What the agent learns is plain text and code that the business's own experts can read, edit and veto; it updates when an SOP changes, and it carries to the next model the day that model ships. A harness learned on a small model made a frontier model better with no learning of its own, and on two of three benchmarks each success costs less, not more. The hard part is not the optimizer. It is the Twin and the Verifier, and with them, the last mile becomes something an agent can learn.

References

[1] Zhao, E., Tong, Y., et al. BizAgentBench: Unveiling Hidden Challenges in Deploying Long-Horizon Agents for Real-World Business. arXiv:2610.20424, 2026.

[2] Shi, Q., Zytek, A., Razavi, P., Narasimhan, K., and Barres, V. τ-Knowledge: Evaluating Conversational Agents over Unstructured Knowledge. arXiv:2603.04370, 2026. (Its banking domain is listed on the τ leaderboard as τ^3-Banking.)

[3] Malay, S. K. R., et al. EnterpriseOps-Gym: Environments and Evaluations for Stateful Agentic Planning and Tool Use in Enterprise Settings. arXiv:2603.13594; ICML 2026.

[4] Rayo. Introducing Rayo: Teach AI Agents Your Business. https://www.rayo.com/blog/, 2026. (Financial Operations Benchmark: https://www.rayo.com/#benchmark.)

[5] Tu, V., and Jain, A. Introducing FrontierCUA: Hard but Fair Benchmark for Evaluating Enterprise Computer Use. Collinear AI blog, November 2026. https://blog.collinear.ai/p/frontiercua

[6] Krentsel, A., Ratnasamy, S., Stoica, I., Casado, M., Goyal, A. Exo: An Architecture for Self-Improving Systems at Runtime. arXiv:2611.00463, 2026.

[7] Pu, L., Li, X., Liu, Y., Cao, T., and Yang, B. Explore More, Drift Less: Outcome-Only Reinforcement Learning Can Suffice for Long-Horizon Interactive Agents (CANOPY). arXiv:2609.01245, 2026.

[8] Szot, A., et al. Scaling Laws for Tool-Use Agents via Controllable Environment Synthesis (SynthAPI). arXiv:2610.37648, 2026; AppWorld leaderboard submission (Qwen 3.5 27B).

[9] Qwen Team, Alibaba. Release announcements: Qwen3.5 (February 2026), Qwen3.6 (April 2026), Qwen3.7 (May 2026), Qwen3.8 (August 2026) and Qwen4-Max (November 2026). https://qwen.ai/blog?id=qwen3.5; https://qwen.ai/blog?id=qwen3.6-27b; https://www.alibabacloud.com/blog/qwen3-7-the-agent-frontier_603154; https://www.alibabacloud.com/blog/alibaba-unveils-qwen3-8-27b-and-releases-weights-of-qwen3-8-flagship-model_603463; https://www.alibabacloud.com/press-room/alibaba-unveils-qwen4-max

[10] Z.ai. Release announcements: GLM-4.7 (December 2025), GLM-5 (February 2026), GLM-5.1 (April 2026), GLM-5.2 (June 2026) and GLM-5.3 (August 2026). https://z.ai/blog/glm-4.7; https://z.ai/blog/glm-5; https://z.ai/blog/glm-5.1; https://z.ai/blog/glm-5.2; https://z.ai/blog/glm-5.3

[11] Agrawal, L. A., et al. GEPA: Reflective Prompt Evolution Can Outperform Reinforcement Learning. arXiv:2507.19457, 2025; ICLR 2026.

[12] Zhang, Q., et al. Agentic Context Engineering: Evolving Contexts for Self-Improving Language Models. arXiv:2510.04618; ICLR 2026.

[13] Yang, Y., et al. SkillOpt: Executive Strategy for Self-Evolving Agent Skills. arXiv:2605.23904, 2026; NeurIPS 2026.

[14] Ni, J., et al. Trace2Skill: Distill Trajectory-Local Lessons into Transferable Agent Skills. arXiv:2603.25158, 2026.

[15] Ma, Y., Huang, Y., Bao, H., Zhuang, H., Shukla, S., Galley, M., Zhang, X., Feuerriegel, S. SkillGen: Verified Inference-Time Agent Skill Synthesis. arXiv:2605.10999, 2026; NeurIPS 2026.

[16] Lee, Y., Nair, R., Zhang, Q., Lee, K., Khattab, O., Finn, C. Meta-Harness: End-to-End Optimization of Model Harnesses. arXiv:2603.28052, 2026.

[17] Zhou, S., et al. AgenticFinBench: Benchmarking Financial Agents on Diverse Tasks from Real-World Requests. arXiv:2611.04813, 2026.

[18] Ross, S., Gordon, G., Bagnell, J. A. A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning (DAgger). arXiv:1011.0686; AISTATS 2011.

[19] Agarwal, R., Vieillard, N., Zhou, Y., Stanczyk, P., Ramos Garea, S., Geist, M., Bachem, O. On-Policy Distillation of Language Models: Learning from Self-Generated Mistakes (GKD). arXiv:2306.13649; ICLR 2024.

[20] Lu, K., et al. On-Policy Distillation. Thinking Machines Lab, Connectionism blog, October 27, 2025.

[21] Ross, S., Bagnell, J. A. Efficient Reductions for Imitation Learning. AISTATS 2010.

[22] Snell, C., Klein, D., Zhong, R. Learning by Distilling Context. arXiv:2209.15189, 2022.

[23] Konda, V. R., Tsitsiklis, J. N. Actor-Critic Algorithms. NeurIPS 1999.

[24] Mnih, V., et al. Asynchronous Methods for Deep Reinforcement Learning (A3C/A2C). arXiv:1602.01783; ICML 2016.

[25] Gao, L., Schulman, J., Hilton, J. Scaling Laws for Reward Model Overoptimization. arXiv:2210.10760; ICML 2023.

[26] Wang, Z. Z., Gandhi, A., Neubig, G., Fried, D. Inducing Programmatic Skills for Agentic Tasks (Agent Skill Induction). arXiv:2504.06821; COLM 2025.

[27] Zheng, B., Fatemi, M. Y., Jin, X., Wang, Z. Z., Gandhi, A., Song, Y., Gu, Y., Srinivasa, J., Liu, G., Neubig, G., Su, Y. SkillWeaver: Web Agents can Self-Improve by Discovering and Honing Skills. arXiv:2504.07079, 2025.

[28] El Yadouni, S., Li, G. TraceCompiler: Skill-Guided Mining and Compilation of LLM Agent Traces into Mostly Deterministic Workflows. arXiv:2608.02680, 2026.

[29] Huang, K.-H., et al. CRMArena-Pro: Holistic Assessment of LLM Agents Across Diverse Business Scenarios and Interactions. arXiv:2505.18878; TMLR 2026.

[30] Foody, B., Vidgen, B., Robinson, I. Expert Data Drives Model Performance. Mercor blog, January 23, 2026. https://mercor.com/blog/expert-data-drives-model-performance

[31] Grupen, N., Pereyra, J., Pereyra, G., Garg, R., Phillips, J., Feng, R. Training a Legal Agent With Applied Compute. Harvey blog, June 22, 2026. https://harvey.ai/blog/training-a-legal-agent-with-applied-compute

[32] Khattab, O., et al. DSPy: Compiling Declarative Language Model Calls into State-of-the-Art Pipelines. arXiv:2310.03714; ICLR 2024.

[33] Yuksekgonul, M., et al. Optimizing Generative AI by Backpropagating Language Model Feedback (TextGrad). Nature 639, 609–616, 2025; arXiv:2406.07496.

[34] Wang, Z. Z., Mao, J., Fried, D., Neubig, G. Agent Workflow Memory. arXiv:2409.07429; ICML 2025.

[35] Applied Compute, 2026b. Remember, Refine, Retrieve: A Context Engine for Enterprise Agents. May 1, 2026. https://appliedcompute.com/research/remember-refine-retrieve

[36] Yao, S., Shinn, N., Razavi, P., Narasimhan, K. τ-bench: A Benchmark for Tool-Agent-User Interaction in Real-World Domains. arXiv:2406.12045; ICLR 2025.

[37] Barres, V., Dong, H., Ray, S., Si, X., Narasimhan, K. τ^2-Bench: Evaluating Conversational Agents in a Dual-Control Environment. arXiv:2506.07982; ICML 2026.

[38] Trivedi, H., et al. AppWorld: A Controllable World of Apps and People for Benchmarking Interactive Coding Agents. arXiv:2407.18901; ACL 2024.

[39] Zhu, Y., et al. Establishing Best Practices in Building Rigorous Agentic Benchmarks. arXiv:2507.02825; NeurIPS 2025 (Datasets and Benchmarks Track).

[40] Skalse, J., Howe, N. H. R., Krasheninnikov, D., Krueger, D. Defining and Characterizing Reward Hacking. arXiv:2209.13085; NeurIPS 2022.

[41] Applied Compute, 2026a. Building State-of-the-Art Agents with Mercor. Case study, https://www.appliedcompute.com/case-studies/mercor.

[42] Islam, P., et al. FinanceBench: A New Benchmark for Financial Question Answering. arXiv:2311.11944, 2023.

[43] Bigeard, A., Nashold, L., Krishnan, R., Wu, S. Finance Agent Benchmark: Benchmarking LLMs on Real-world Financial Research Tasks. arXiv:2508.00828, 2025.

[44] Pauli, W. M., Panda, S., Admassu, K., Bleik, S., Okerinde, A., Reynolds, J. FORCE-Bench: A Benchmark, Dataset, and Evaluation Harness for Agentic AI in Enterprise Finance. arXiv:2607.19409, 2026.

[45] Zheng, L., et al. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. arXiv:2306.05685; NeurIPS 2023 (Datasets and Benchmarks Track).

[46] Wang, P., et al. Large Language Models are not Fair Evaluators. arXiv:2305.17926; ACL 2024.

[47] Panickssery, A., Bowman, S. R., Feng, S. LLM Evaluators Recognize and Favor Their Own Generations. arXiv:2404.13076; NeurIPS 2024.

[48] Lù, X. H., et al. AgentRewardBench: Evaluating Automatic Evaluations of Web Agent Trajectories. arXiv:2504.08942; COLM 2025.

[49] Zhuge, M., et al. Agent-as-a-Judge: Evaluate Agents with Agents. arXiv:2410.10934; ICML 2025.

[50] Uesato, J., et al. Solving Math Word Problems with Process- and Outcome-Based Feedback. arXiv:2211.14275, 2022.

[51] Lightman, H., et al. Let's Verify Step by Step. arXiv:2305.20050; ICLR 2024.

[52] Cobbe, K., et al. Training Verifiers to Solve Math Word Problems. arXiv:2110.14168, 2021.

A Benchmarks and evaluation protocol

Table 8 summarizes the public benchmarks, and Table 9 how each result was produced, including what the harness was exposed to during learning.

Table 8: Public benchmarks. Each baseline is the benchmark's default agent with no learned harness.

Benchmark	What it tests	Tasks	Agent model (frozen)	Success	Trials
AppWorld	autonomous work across 9 apps, no user to ask	732	Qwen 3.5 9B; Qwen 3.5 27B; Claude Haiku 5.5	unit tests (TGC, SGC)	3
τ^3-Banking	policy from a 698-document knowledge base, with a customer	97	Claude Opus 5; Claude Haiku 5.5	database-state reward	4
BizAgentBench	following and enforcing payments SOPs	50	Claude Haiku 5.5; GPT-6 Sol	goal, process, protection (strict = all three)	1

Table 9: Evaluation protocol. "Exposure" lists everything the harness or the Verifier learned from or was tuned on.

Benchmark	Exposure during learning	Evaluated on	Simulator	Grader	Trials
Financial Ops	experts' recorded training cases; synthetic cases	1,000 held-out real cases, untouched	none	domain expert	3
AppWorld	train, dev and synthetic tasks	test_normal, test_ challenge	none	unit tests	3
τ^3-Banking	training tasks (60/40 split): leaderboard trajectories; synthetic tasks	all 97 tasks (Table 4); held-out tasks (Table 5)	GPT-5.2	tau v1.0.1 database state	4
BizAgentBench	10 tasks read; 150 synthetic tasks	all 50 tasks, 40 never read	Claude Opus 5.5	goal, process, protection	1

B Comparison with other harness-learning methods: setup differences

Table 5 compares systems, each used as designed, so they differ in more than their learning signal (a task reward with the optimizer's reflection for SkillOpt, grader feedback that lists the expected actions with the agent's own reflection for Exo, a calibrated Verifier that names the first mistake and its repair for ours). Ours also drew on reference trajectories and could edit scripts and the runtime as well as text; these are part of how the method works, but they mean the comparison does not isolate the signal. The setups differ too: the plain and Rayo arms use 4 trials and the GPT-5.2 customer, SkillOpt and Exo 1 trial and GPT-5.4; the Haiku Exo runs predate a fix for parallel tool calls, which understates Exo; and Exo was free to keep adapting on the held-out tasks, as its protocol allows, while the Rayo and SkillOpt harnesses were frozen. On Claude Opus 5.5, Exo with code editing reaches 61.1% against 52.8% for the plain agent, a difference that is not significant (5 tasks won, 2 lost).

For the held-out tasks with the closest training counterparts, Exo had passed only a few of those counterparts during training, and over training its pass rate stayed flat. These observations are consistent with the learning signal mattering, but the comparison cannot establish it; that needs an ablation that holds inputs, edit scope and setup fixed.

C Memorization audit

We scanned the final harnesses for τ^3-Banking and BizAgentBench, everything the agent can see (guidance, reference files, product data and helper code) plus the runtime, for anything specific to the benchmarks' tasks or customers (Table 10). Neither contains anything from a test task.

Table 10: Memorization audit of the final harnesses. Each check searches the harness for values that could only come from a specific task or customer.

Check	τ^3-Banking	BizAgentBench
Task and customer identifiers: ids, names, contact details, account and record numbers	0 of 1,112 values from all 17 bank tables	0 of 892 case-specific strings from the 50 private case files
Task references: task ids, or a customer named in prose	0	0
Strings in both the tasks and the harness but not in the shared documents	9 of 3,249, all API vocabulary: tool names such as `apply_for_credit_card`, the argument `new_email`, the status `CANCELLED`	8, all field names and values from records the agent reads in every episode, such as `sum_at_least`; each spans 7 to 14 cases, and the plain agent sees them too
Rare numbers: in at most 3 tasks and in the harness, but not verbatim in the documents	5, all generic: product minimums written without commas, an example amount in a code comment, a formula constant	1, a balance in a worked example in the helper documentation, taken from one development case; no test case contributes any number

Both harnesses hold procedure and facts from the shared knowledge base that every task can read: eligibility rules, fee and interest tables, dispute and payment steps, and checks such as verifying a customer's identity before acting. The only task-derived value is one number in a worked example that shows how to call a helper; it comes from a development case and encodes no answer. No identifier, name or number from a test task appears in either harness, consistent with the rule that held-out tasks never inform a change.

D Cost efficiency

We measure cost per successful task:

$$\text{cost per success} = \frac{\text{agent spend summed over every run attempted, failed or not}}{\text{number of successful runs}}.$$

Agent spend is the agent's actual model spend, including calls made through the harness's internal helper tools, which τ-Bench's recorded trajectories do not show; it excludes the customer simulator, which is the same in both arms.

Per success. On AppWorld, priced at Qwen 3.5 27B rates, a success costs \$0.067 with the harness against \$0.125 without ($\times$0.54). On τ^3-Banking, Claude Haiku 5.5 costs \$0.0234 per trial alone and \$0.0258 with the harness, the figure in our leaderboard submission; because three times as many trials succeed, a success costs \$0.032 against \$0.088 ($\times$0.36). τ-Bench's own cost field records \$0.0176 for the harness because it sees only the calls in the recorded trajectory. The saving comes from failure: failed runs are the longest and most expensive, the harness removes most of them, and each remaining run is more direct. The guidance itself adds cost to every call, which is why a single trial costs slightly more with the harness.

The exception: BizAgentBench. Here a success costs 1.2$\times$ as much with the harness. Its guidance is long (about 42K tokens) and reread on every turn of a long conversation with the customer, and success rises less (30% to 55% strict), so fewer failed runs are removed to offset it. Prompt caching serves 94% of input tokens, which holds the cost increase to 1.2$\times$ even though input tokens per success rise 1.7$\times$. Loading guidance only when a step needs it, rather than on every turn, is the obvious next step.

Learning cost. Building the Twin, calibrating the Verifier and learning the harness is a one-time cost, paid once per workflow and not per run. Excluding the test runs, it cost about \$200 in model spend on AppWorld, \$200 on τ^3-Banking and \$250 on BizAgentBench, and in each case learning took under four hours of run time. At the savings per success above, the learning cost is recovered after about 3,400 successful AppWorld tasks and about 3,600 successful τ^3-Banking conversations. In enterprise work the larger saving is elsewhere: every case the agent completes is one an expert does not have to handle.

E Extended related work

Weight training for agents. RL on agent environments now leads benchmarks such as AppWorld, through outcome-only RL [7] and RL on synthesized environments [8]. In professional domains, RL on expert-built tasks nearly doubled pass@1 across law, banking and consulting [30], and Harvey validated its graders and improved its harness before RL, with the combination raising its legal agent's scores [31]. These methods learn on-policy but store in weights.

Prompt, context and harness optimization. DSPy compiles and optimizes language-model pipelines [32], and TextGrad propagates textual feedback [33]. GEPA evolves prompts by reflecting on trajectories and keeps a Pareto front of candidates [11]; ACE grows a context playbook by generating, reflecting and curating [12]; Meta-Harness searches over harness code with an agentic proposer [16]; and Exo lets an agent edit its own harness at runtime, mainly cutting cost at a saturated success rate [6]. Its AppWorld study does not use the benchmark's train and test splits: it learns on one variant of each of 50 scenarios and is evaluated on other variants of the same scenarios, so its results are not comparable with test-split numbers.

Learning skills from experience. The closest line of work learns natural-language skills for a frozen model from its own trajectories. SkillOpt treats a skill document as trainable: a frontier optimizer turns scored trajectories into bounded edits, and a held-out gate accepts only edits that improve validation performance [13]. Trace2Skill analyzes many trajectories in parallel and consolidates their lessons hierarchically into one skill that transfers across models [14]. SkillGen contrasts successful and failed trajectories and verifies each skill as an intervention, running the same tasks with and without it [15]. Agent Workflow Memory extracts recurring workflows from experience [34]. A second line learns executable skills: Agent Skill Induction synthesizes, verifies and reuses programmatic skills [26]; SkillWeaver has web agents explore a site, write skills as APIs, and practise to refine them [27]; and TraceCompiler mines tool traces into mostly deterministic workflows [28]. Context Engine refines documents and traces into a retrievable context base [35]. Our edit-and-gate step is close to SkillOpt's bounded edits with a held-out gate and to SkillGen's intervention test, and our tools play the role of programmatic skills. What we add is the source of the signal: a Verifier, calibrated on ground truth like a critic, that diagnoses the first mistake at the agent's own states, calibrated against a Twin so that learning works where no benchmark grader or success label exists, and evaluated on real enterprise cases.

Environments and benchmarks. Hand-built environments [36; 37; 38] are faithful but small, and some can be passed by trivial agents [39]; synthetic generation scales [8] but rewards the generator's mistakes without a strong check [40; 41]. Finance and enterprise benchmarks [1; 3; 17; 29; 42; 43; 44] show large gaps on procedural work.

LLM judges and verifiers. LLM judges agree well with people on open-ended outputs [45] but are biased by order and toward their own outputs [46; 47], and no single judge is reliable across agent benchmarks [48]. Agent-as-a-Judge evaluates intermediate steps [49]; process supervision rewards each step [50; 51]; trained verifiers rank solutions [52]. A learned judge used as a reward can be over-optimized, improving the proxy while the true objective degrades [25]. We calibrate the Verifier on the issues it names and on whether its repairs succeed, because a judge that is right about the outcome for the wrong reason sends the edit to the wrong place.

F Where the method fits

The method depends on how much of the truth can be checked automatically. Let

$$C_g = \frac{N_{\text{grounded automatically}}}{N_{\text{labels required}}}, \qquad C_h = \frac{N_{\text{grounded by a person}}}{N_{\text{labels required}}} \tag{6}$$

be the share of required labels grounded automatically (by systems, rules or recorded outcomes) and the share that needs a human expert. Operational work in finance and enterprise systems usually has high C_g: a ledger balances or it does not, an approval exists or it does not, a case is closed with the right status or it is not. Workflows with low C_g turn into expensive human labeling and are a poorer fit.

G RL and on-policy distillation updates

For reference, the two weight-space updates that §2.2 contrasts with harness training are

$$\text{RL:} \quad \theta \leftarrow \theta + \eta \, \nabla_\theta \, \mathbb{E}_{\tau \sim p_\theta} \big[R(\tau) \big] \tag{7}$$

$$\text{On-policy distillation:} \quad \theta \leftarrow \theta - \eta \, \nabla_\theta \, \mathbb{E}_{\tau \sim p_\theta} \sum_i \mathrm{KL}\big(\pi_{\text{teacher}}(\cdot \mid s_i) \,\big\|\, \pi_\theta(\cdot \mid s_i) \big) \tag{8}$$

where θ are the model's weights, p_θ the distribution over executions they induce, η a learning rate, $R(\tau)$ the reward for an execution, s_i the i-th state of the execution, $\pi_\theta(\cdot \mid s_i)$ and $\pi_{\text{teacher}}(\cdot \mid s_i)$ the student's and the teacher's distributions over the next action there, and KL the Kullback–Leibler divergence between them. Both update θ. Harness training computes neither: it keeps their on-policy sampling, replaces the reward or the teacher's distribution with the Verifier's diagnosis (2), and replaces the gradient step with the accepted edit (4).