THE GRADER IS AN INTERVENTION: EVALUATOR NON-INTERFERENCE FOR STATEFUL AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Benchmarks for AI agents score an agent by checking the environment it leaves behind, assuming the check changes nothing. Training, search and early stopping now grade episodes in progress or more than once. We show that a grader can score the current state correctly and still change a later score. We call the requirement that extra grading leave recorded outcomes unchanged evaluator non-interference, and test it in shipped code. Computing τ-bench's reward writes the correct answer into the environment, so a public training wrapper that grades twice reports 127 passes where the benchmark reports 22. On AppWorld, grading after every API call turns 17 of 142 published successes into failures, and the benchmark's own analysis script destroys 8. There, one grading call overwrites shared state in two ways, and undoing each repairs exactly its own cases, even on held-out tasks. With WebArena's original evaluator, though not its PyPI port, BrowserGym's adapter lets the grader move the agent's page. In a disposable copy of the environment's process, grading changes no replayed outcome beyond run-to-run noise. Grading is therefore part of the experiment, and benchmarks should state when their grader runs and what reads the state it writes.

1 INTRODUCTION

Benchmarks for stateful agents, which act on an environment that keeps state such as a database, a file system or a web page, score an agent by running the environment's own tests against the state it leaves behind. We call the code that runs these tests the grader; benchmarks also call it an evaluator or a reward function. Its call is treated as a passive read: the grader looks, and what it looks at is assumed not to move. Several ways of using agents now depend on this assumption, because they grade an episode, one attempt at a task, while it is still running or more than once. Step-level rewards for training (Yuan et al., 2026), search over partial solutions (Koh et al., 2024) and early stopping all do so, and AppWorld's documentation invites it: one "can run the above at any point", and this "may be useful for RL rewards" (Trivedi et al., 2024b).

The assumption can fail without anyone noticing, because a grader can score its input correctly and still change it. τ-bench's reward function checks an episode by rebuilding the correct end state on the live environment, reloading the initial database and replaying the task's reference solution. A first call therefore scores a failed episode correctly but leaves the answer in the environment, and a second call on the same episode can find it and report a pass. AppWorld's grader swaps in its own bookkeeping for a component the running environment shares, so after one call the environment tracks the agent's later work in the grader's books rather than its own. In both cases the first score is correct. Whether a later score changes depends on when the grader runs and on what afterwards reads the state it wrote, the agent's next action or another grading call. Grading is then an intervention on the experiment, and this is the explanation the paper tests.

We define evaluator non-interference, the requirement that extra grading leave an episode's recorded outcome unchanged (§2). To measure it we run each episode with and without extra grading, and we repeat the ungraded run to see how much outcomes vary on their own, a floor against which every difference is read. We then test code that benchmarks and their users ship (§3, Figure 1). A public τ-bench reinforcement-learning environment grades a second time after the benchmark has

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.

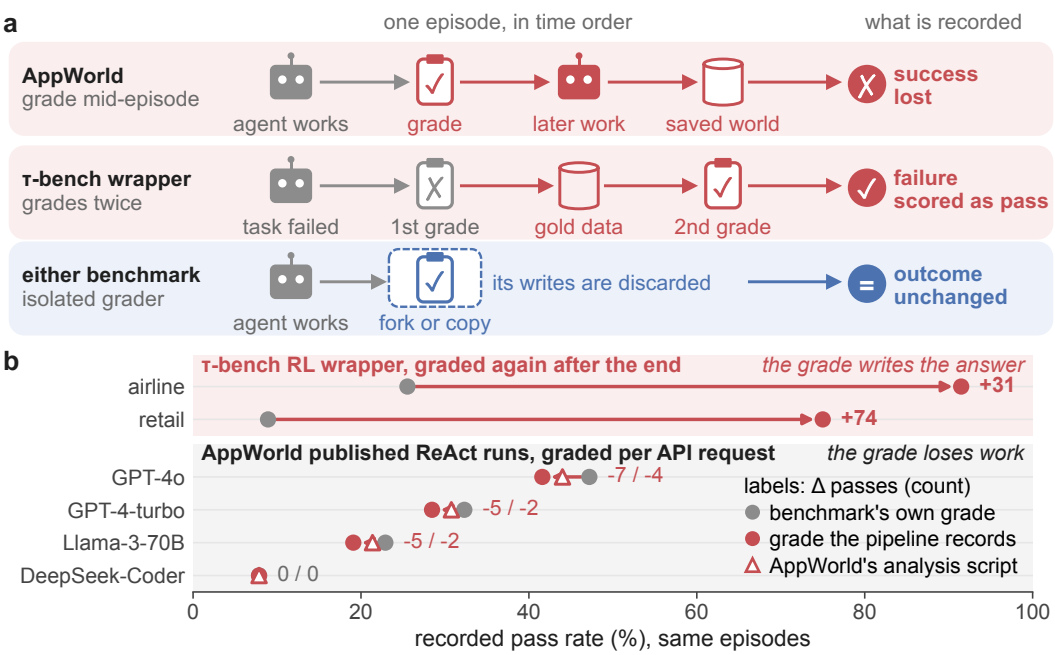

Figure 1: **A correct grade can change the outcome recorded next. a,** Three episodes in time order. On AppWorld the agent's later work is recorded in the grader's books and missing from the saved world. The public τ-bench wrapper grades twice, and the second grade reads the correct answer, the gold data, that the first wrote. A grader kept apart from the environment keeps its writes to itself. **b,** Recorded pass rates on the same episodes, per τ-bench domain (47 and 112 episodes) and per published AppWorld model (125 to 140 eligible tasks). Labels give the change in passes. For AppWorld the first number is grading after every API call and the second its analysis script.

ended the episode. Run as shipped on all 165 tasks, it reports 127 of 159 comparable episodes as passes where the benchmark's own grade passes 22. Grading AppWorld's published agent runs after every API call destroys 17 of 142 published successes, and the benchmark's own analysis script destroys 8. On WebArena, BrowserGym's adapter hands the grader the page the agent is using, and WebArena's upstream evaluator, the original implementation, navigates it after every step. In a live run 92% of the agent's observations are then the grader's target page. That disturbs the state the agent acts from, a second property, non-disturbance, that we state separately.

We then ask why (§4). On AppWorld the changes appear in three independent setups, including the benchmark's own server and a live agent, so they are not an artifact of our measurement code. In our trajectories their number grows with how often the grader runs, and no episode changed when at most two actions followed the call. A single grading call corrupts the record in two separate ways, one hiding work the agent saved and one losing work it does later. Undoing each repairs exactly its own cases, here and on 130 tasks from a held-out split. Finally we ask what protects a pipeline (§5). Running the grader in a disposable copy of its process, a fork, keeps its writes out of the agent's memory except through files, under a condition we check rather than assume. Against an ungraded replay it changes no outcome beyond the floor, at a cost we report. Reusing the first grade repairs the τ-bench pipeline, and we say which leaks neither fix closes.

2 A CONTRACT FOR GRADING AND ITS MEASUREMENT

An episode applies actions $a_1, \ldots, a_n$ to an environment and leaves a terminal state behind. The benchmark turns that state into a number by running its own tests over it. Write $R(G)$ for the outcome it records when the grader is additionally called at the positions in a schedule G, and $R(\varnothing)$ for the outcome with no additional grading. A schedule may include calls after the episode ends but before its score is recorded.

Definition 1 (Evaluator non-interference)**.** A grader is non-interfering when extra calls never change the recorded outcome:

$$R(G) \;=\; R(\varnothing) \qquad \text{for every schedule } G. \tag{1}$$

When non-interference fails, the number a pipeline reports is $R(G)$ while its readers take it for $R(\varnothing)$, and the gap comes from the pipeline, not the agent. We compare recorded outcomes, not states, since that is what leaderboards publish and training loops consume. The agent never sees the extra grades, so a difference is a side effect, not a response to reward.

Two properties, not one. Equation (1) is about the recorded outcome. A grader can also change the state the agent acts from next, by navigating its page or rewriting its data, without changing any recorded score. We call leaving that state alone non-disturbance, and the two can fail separately. BrowserGym's adapter, with WebArena's upstream evaluator, moves the agent's page, yet on our 50 test pages it returns the same grade as a check on a separate page. τ-bench's grader changes the recorded outcome, and AppWorld's fails both (§4.1).

Conditions for interference. Three things must hold at once. The grader must act on state the environment also holds, in the same process or through a shared file, database or browser. It must write state that outlives the call, such as a module-level cache or a class-level binding. That state must then be read back, by the environment as it continues or by a later grading call. If any one fails, grading cannot reach a later outcome. The third cannot be seen in an API's signature, since it depends on what runs later. Grading only at the end avoids later actions but not necessarily a later reader, since a second grading call can read what the first wrote. A grader can therefore score its input correctly and still corrupt the pipeline's result.

The census. Equation (1) compares two runs, and we perform both. We replay one fixed action sequence under four arms that differ only in the grader. An ungraded arm measures $R(\varnothing)$, and an in-process arm measures $R(G)$ as shipped code runs it. A forked arm makes the same calls in a discarded child process, so a change that appears in process but not in the fork must come from sharing the process. A repeat of the ungraded arm differs in nothing, so it measures the environment's own noise, and we read every difference against it.

On AppWorld we replay one task from each of 55 task families. Three models recorded one trajectory per family, and each model's recordings form a trajectory set, so the 55 families under three sets give 165 rows. A cell is one row under one grading schedule, run in each arm. The floor, the repeat's disagreement with the first run, is zero on all 165 rows. A row is eligible when its ungraded and forked arms both reproduce the recorded trajectory, as 151 do. A changed in-process outcome on an eligible row is an interference event (Appendix B). Live-agent runs let the agent choose its actions, with a repeat of one arm as the floor. Table 1 maps every experiment.

Table 1: **The experiments, their units and what each compares.** Each count compares two arms on the same units, except the repair run, which regenerates episodes. The floor is the disagreement between two runs that differ in nothing.

experiment and units (where reported)	comparison	floor or check
τ-bench wrapper: 159 episodes (Table 2)	second vs benchmark grade	repair run: data unchanged
τ-bench closed loop: 165 tasks (Fig. 2a)	in process vs forked, 4 setups	repeats differ on 4–36
AppWorld published runs: 526 tasks (Fig. 1b)	added grading vs published	forked 2, repeat 2 of 666
BrowserGym live: 66 tasks, 64 paired (Fig. 2b)	agent's vs disposable page	isolated arms differ on 0–2
Census, 151 rows; HTTP, 54 families (Table 3)	in process vs ungraded, forked	floor 0
Live ACE, two models: 55 families (Table 3)	in process vs isolated	floors 2 and 0 of 55
Cadence and placement: 825 cells each (Fig. 3)	in process vs forked	forked 0 of 825
Restoration: 18 damaged rows (Fig. 4b)	restore a subset of 7 attributes	two runs agree

Table 2: **The public τ-bench pipeline, end to end as shipped and with the repaired rubric.** The benchmark's grade is what its own step computed when the episode ended; the pipeline's is the rubric's second call. The repair run regenerates trajectories, so its counts are its own; one of its airline episodes stopped on a model-service error.

	airline	retail	both
episodes with the benchmark's own grade (of 50 and 115)	47	112	159
passes by the benchmark's own grade	12	10	22
passes reported by the pipeline	43	84	127
excess passes, benchmark failures reported as passes	31	74	105
on tasks without an output check	31	69	100
on tasks whose output check the agent met	0	5	5
passes the pipeline turns into failures	0	0	0
fresh repair run: ended episodes returning the cached grade	48	113	161
fresh repair run: step-capped, graded once on a copy (passes)	1 (0)	2 (0)	3 (0)

3 Interference in shipped pipelines

We first ask whether shipped code breaks the contract. In τ-bench and AppWorld it does, and in BrowserGym, with WebArena's upstream evaluator, the grader moves the page the agent sees next.

3.1 A public τ-bench pipeline grades twice

τ-bench (Yao et al., 2024) exposes its reward as a method on the live environment. The method reloads the initial data, replays the task's ground-truth actions on that same object, and never restores the agent's data (`envs/base.py:133-136`). The benchmark calls it once, in the step that ends an episode, and we found no pipeline that calls it earlier. We found one public reinforcement-learning environment that calls it a second time after that step (Prime Intellect community, 2026). Its scoring function, the rubric, calls the reward method again on the same object, so the second call reads the answer the first wrote.

We ran that environment as shipped, through its own evaluation command, on all 165 test tasks. Llama-3.3-70B-Instruct plays agent and simulated user. The agent samples at temperature 0, and the user at the server's default of 1, because τ-bench's user simulator passes no temperature. We add only a recorder of the reward the benchmark's own step computes (Table 2; Appendix C.1). Six episodes hit the step cap and have no grade from the benchmark. On the other 159 the benchmark passes 22 and the pipeline reports 127. The benchmark's low pass rate reflects the stochastic user; the comparison pairs the two grades of each episode.

All 105 excess passes are tasks the benchmark failed on its database check, and no pass becomes a failure. Our protocol predicted that every excess pass would fall on a task without an output check, the part of the reward that looks for expected strings in the agent's replies. Of the 105, 100 do and 5 do not, so that prediction failed. The 5 are retail tasks whose expected outputs the agent had stated. A diagnostic made afterwards, with no model call, says why. Whether the agent's messages contain the expected outputs reproduces the pipeline's verdict on all 159 episodes. After the first call, the second no longer tests whether the agent changed the database.

The smallest repair is to reuse the first grade, which agrees with the benchmark by construction. We ran it in the same pipeline on new episodes to check the plumbing. All 161 episodes that ended returned their cached grade, three step-capped episodes were graded once on a copy, and one episode stopped on a model-service error. The rubric's call never changed the original data.

Grading before the end errs the same way. In our own closed-loop setup, where a live agent talks to a simulated user, we ran all 165 tasks. Grading in process after every step adds between 29 and 79 passes across four configurations, from 42 to 117 for Qwen3-14B (Figure 2a; Appendix D). Runs of one arm differ on 4 to 36 tasks, so these contrasts give size and direction, not a rate to expect

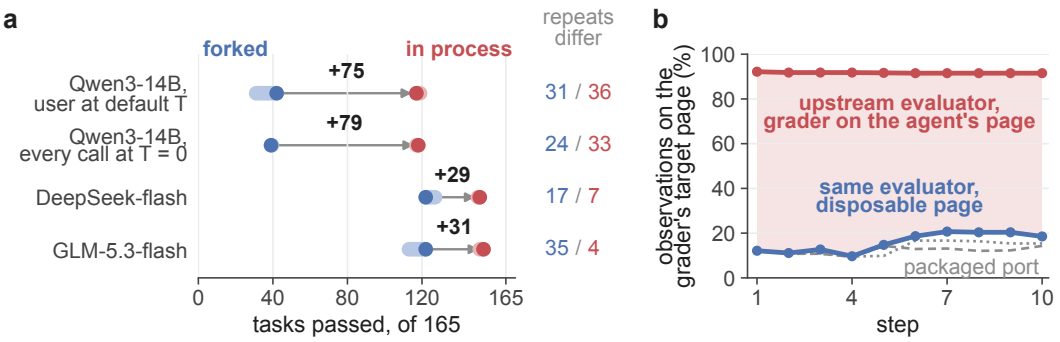

Figure 2: **With live agents, in-process grading inflates τ-bench passes, and WebArena's upstream evaluator puts BrowserGym's agent on the grader's page. a,** τ-bench in closed loop on 165 tasks, four agent and user configurations. Bars span two runs' pass counts, arrows give the shift between first runs, and numbers at right count the tasks on which an arm's two runs disagree, blue forked and red in process; T is the sampling temperature. **b,** BrowserGym on 66 WebArena `shopping_admin` tasks, 64 paired under the upstream evaluator: share of observations at each step, among episodes still running, that are the grader's target page. On a disposable page only the agent's own visits count. Grey, the packaged port, where both arms use a disposable page.

elsewhere. In an exploratory replay, forked calls change nothing (Appendix C.2). The successor, τ^2-bench (Barres et al., 2025), grades in a freshly built environment and closes this channel.

3.2 AppWorld's published runs and its analysis script

Rebuilt published trajectories. AppWorld's (Trivedi et al., 2024a) public `evaluate` method, served over HTTP beside the action endpoint, grades a live world from inside it, whereas its offline batch evaluator refuses to start while a world is open. The benchmark ships its paper's baseline runs, with each agent's API requests and per-task evaluations. Those runs graded once at the end, so we ask what grading after every action would record on the same trajectories. We rebuilt the four published ReAct runs on `test_normal`, one request per action (Appendix E, Figure 9). Of the 672 published evaluations, 666 are rebuilt, 662 reproduce the published evaluation, and 526 also pass the census's eligibility. Three of the four runs fall below the 80% eligibility our protocol required. On those 526, grading after every action changes 36 evaluations, destroys 17 of the 142 published successes and manufactures none. Because eligibility already requires the forked arm to agree, the independent control uses all 666 rebuilt tasks. There in-process grading changes 45 evaluations, while the forked arm and a repeated ungraded replay each differ from the ungraded arm on 2, and none of those 4 changes a success. The published logs keep each API request but not the code that issued it, so we grade after every request, a finer schedule than the agent's own steps.

The shipped analysis script. The script (Trivedi et al., 2025) plots score against an interaction limit. For each task it rebuilds the world, replays the interactions five at a time, grades after every block, and saves the replayed world back over the experiment. On the published runs it changes 16 of the 526 eligible evaluations and destroys 8 of the 142 successes. On experiments rebuilt from our own 165 trajectories, at the agents' own interaction cadence, it changes 9, 4 of them successes turned into failures. A copy that moves each grading call into a fork changes none (Appendix F).

Online rewards versus saved-world outcomes. We also replay our own 165 trajectories as a training loop that grades after every step and uses the grade as a reward. Against the forked arm, 5 of the 94 episodes that complete the task lose an earned reward (Figure 5, Appendix B). The same schedule destroys 9 saved successes (§4.2), and for 5 of them the last in-process grade still reports success, so the reward a training loop sees and the score a leaderboard records disagree (Figure 6).

3.3 With WebArena's upstream evaluator, BrowserGym moves the page

BrowserGym's (Le Sellier de Chezelles et al., 2025) adapter for WebArena (Zhou et al., 2024) calls the evaluator after every step and hands it the agent's live page; the official implementation grades

once, at the end (BrowserGym contributors, 2025). WebArena's upstream evaluator, at the commit we pinned and on its main branch, navigates that page on 302 of 812 tasks. On 50 local test pages, BrowserGym's recovery step, which re-attaches to the page but never navigates back, left it where the check put it. The same check on a disposable page gave the same grade in 50 of 50 (Appendix G). The `libwebarena` port that `browsergym-webarena` pins opens a disposable page instead, so PyPI installs are not exposed.

We ran a live agent on 66 `shopping_admin` tasks under both configurations. Under the port both arms grade on a disposable page by construction. Under the upstream evaluator, 92% of the agent's observations after a step are the grader's target page, against 16% on a disposable page (Figure 2b). Outcomes move little. Under the port each arm passes 5 of 66 tasks. Under the upstream evaluator the shipped arm passes 3 of 64 tasks and the disposable page 5, and the two lost tasks pass in all three page-isolated arms, those that grade on a disposable page. Page-isolated arms that should agree differ on 0 to 2 of 66 tasks. The agent also solved too few tasks to pass our competence gate of 10 successes, so we claim the disturbance and not an outcome effect (Appendix G.1).

4 Mechanisms and causal tests on AppWorld

Section 3 shows that outcomes move in shipped pipelines. To see why, we take apart AppWorld, whose world is a single Python process we can replay, instrument and fork.

4.1 The changes survive three setups

Two further setups test whether our own in-process instrumentation creates the changes. Under HTTP, with the server AppWorld ships and only its documented endpoints, we replay one trajectory per family. A control arm issues only `/execute`, an interleaved arm adds `/evaluate` after every `/execute`, and a third arm repeats the control. All are scored after `/close` by the offline evaluator in a fresh process, which is how leaderboard numbers are made. The repeated control reproduces the control on all 54 families that completed every arm, so the floor is zero. Interleaving the two endpoints changes the official evaluation on 7 of the 54 and moves 3 from success to failure. These are exactly the 7 families the in-process census flags, those with a changed outcome on any row, six of them on eligible rows (Appendix A). A fourth arm that snapshots and restores the world around each `/evaluate` changes none but leaves it unclosable (Appendix H).

Replay cannot show an agent reacting to a world the grader has touched. We therefore run the ACE agent (Zhang et al., 2025) at a pinned commit under greedy decoding against stock AppWorld. Its two arms differ only in whether the per-step grading call is isolated, and a second run of the isolated arm is the floor, which disagrees with the first on 2 of 55 families. The agent passes 22 of 55 families isolated and 20 in process. Its graded arm differs on all 7 families the census flags and on 1 of the others, which we call cleared. Two of these eight also changed between the two isolated runs, so they are within the floor (Table 3). Of the seven, three lose a success and none gains one, as under HTTP, and one cleared family gains a success within the floor. These are counts on fixed families, and a family that matched itself once may still vary. All three score with the offline evaluator.

A second model family keeps the direction but not the clean split. With Llama-3.3-70B-Instruct in place of Qwen3-32B, the census flags 4 of the 55 families. The graded arm moves all 4 and 2 of the 51 cleared ones against a floor of 0, destroying 2 successes and manufacturing none. Its protocol, fixed before the run, predicted that no cleared family would move, so that part failed (Appendix B). Replaying the live agent's own actions explains both moves, and both families are flagged on the Qwen trajectories. In one, the second channel of §4.3 acts on actions the census never replayed. In the other, grading changes what one of the agent's actions returns, and the agent then acts differently.

4.2 Grading frequency and placement

A single rate cannot tell whether a few fragile tasks break or the practice itself does damage, so we hold the action sequence fixed and vary the schedule. The grader runs either in the environment's process or in a forked child that is discarded, with the same calls in the same places (Appendix B).

Grading cadence. Changed rows rise with the number of calls (Figure 3a). One call after the last action changes none of the 165 rows, a call after every eighth, fourth or second action changes 8, 10

Table 3: **The families the replay census flags also move over HTTP and under a live agent.** A family is flagged when the replay census changed its outcome on any row, and cleared otherwise; each model's census defines its own grouping. The HTTP client completed every arm on 54 of 55 families and each live agent on all 55. A family is beyond floor when its own repeat did not move.

setup	flagged families move	cleared families move	repeat floor
in-process replay census	*defines the grouping*		0 of 165 rows
black-box HTTP client	7 of 7	0 of 47	0 of 54
live ACE agent, Qwen3-32B	7 of 7 (6 beyond floor)	1 of 48 (0 beyond floor)	2 of 55
live ACE agent, Llama-3.3-70B	4 of 4 (4 beyond floor)	2 of 51 (2 beyond floor)	0 of 55

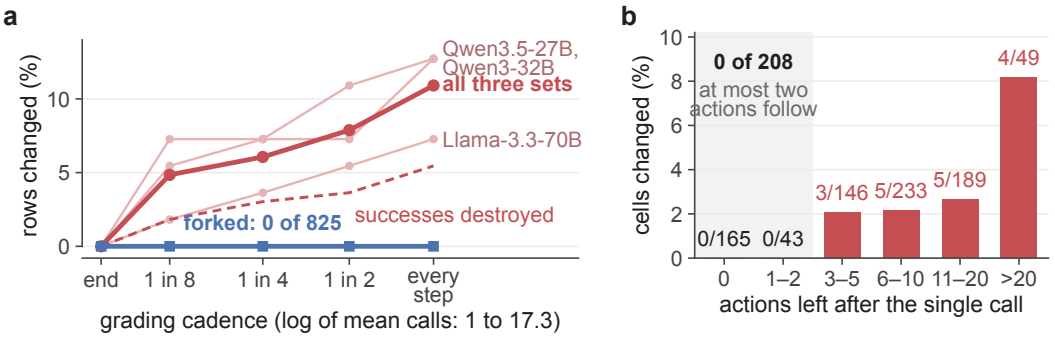

Figure 3: **In these replays, damage grows with how often the grader runs and appears only when later actions remain. a,** Share of rows whose recorded outcome changed, per trajectory set (of 55) and pooled (of 165), as grading becomes more frequent. The x-axis is logarithmic in the mean number of calls per episode, from 1 to 17.3. Dashed, successes destroyed, of 165. Forked calls stay at zero. **b,** One call per episode, grouped by the actions left after it; labels give changed cells over cells, and the 17 changed cells come from seven rows.

or 13, and a call after every action changes 18. Destroyed successes rise through 0, 3, 5, 6 and 9, none is manufactured, and the forked arm changes none of the same 825 cells. These paired counts on fixed trajectories contrast grading once with grading every step and are not a general law.

Placement of a single call. Grading more often is also grading earlier, so a second sweep makes one call and moves it, to after action $\max(1, \lceil fn \rceil)$ of n for $f \in \{0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1\}$. After the last action the call changes none of the 165 rows. The four earlier placements change 3, 3, 4 and 7 rows, too few to rank, and none manufactures a success. Grouped by how many actions follow the call, no cell changes when two or fewer follow, 0 of 208, and 17 of 617 change when more follow (Figure 3b). The 17 come from seven rows, three changed at every non-terminal placement. In this sample the row decides whether a change is possible, and the placement decides only whether it shows. Grading once is therefore not safe. What is safe is grading when nothing that reads the written state remains (§4.3). Our data locate that point only roughly, within the last two actions.

4.3 Two channels opened by one call

The grading call overwrites settings on a class that the running world also uses, and the damage appears only when the world later uses them (Figure 4). The first channel hides saved work: the world keeps its change counts in the grader's books, so the agent's saved work looks unchanged and the evaluator never compares it. The second loses later work: the save that ends every action goes through the grader's change tracker, so work the agent does after the call can go unsaved.

In the source. `evaluate_task` calls `Database.set` on the shared class (`apps/model_lib.py:226`), writing seven attributes it never restores: the counter key `home_path`, the change tracker, two connection handles and three file-name attributes. The counts sit under `home_path` (`collections/models.py:433`), and the tracker is another of the seven (`environment.py:690`).

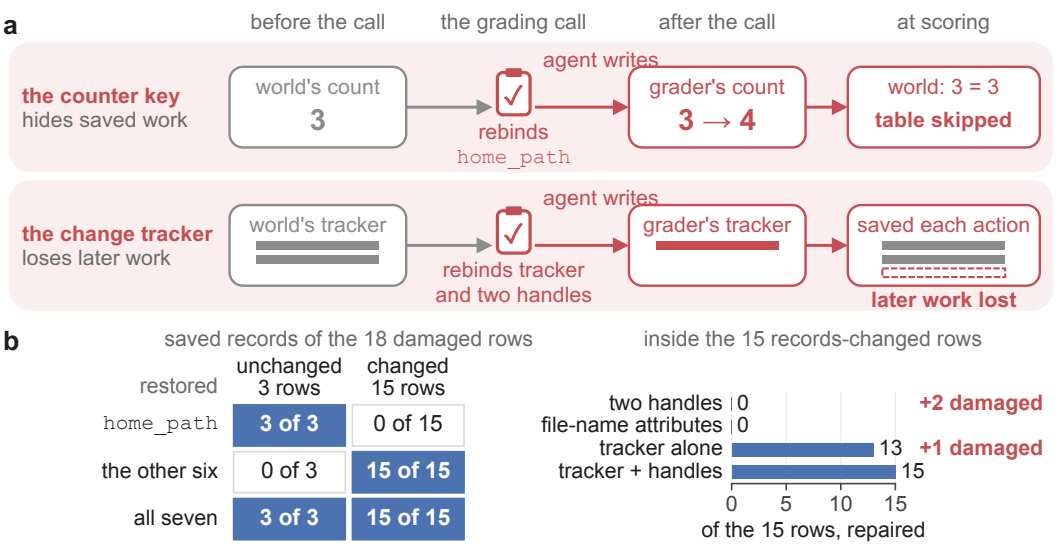

Figure 4: **One grading call opens two disjoint channels. a,** Schematic, with illustrative counts. Through the counter key, the agent's later writes raise the grader's count, the world's count stays put, and the table is skipped unread. Through the change tracker, later writes land in the grader's tracker, and each action's save writes the world without them. **b,** Restorations against the 18 damaged rows, grouped by a signature no restoration uses: whether their saved records changed. The tracker alone damages one row that grading had left intact, and the two handles alone damage two.

Separating them by intervention. Around every grading call we record the seven attributes and restore a chosen subset afterwards, changing nothing else. Grading after every action damages the same 18 of 165 rows in two independent runs. We group them by a signature the restoration does not use: whether the saved records are byte-identical while the grade changes. Restoring `home_path` repairs exactly the 3 rows whose records are unchanged, and the other six attributes repair exactly the other 15 (Figure 4b). Under HTTP no application data file differs for the counter family, while each of the six tracker families shows one (Appendix A). On 130 held-out families, restorations run after a prediction fixed in advance split their five events the same way (Appendix M).

The second channel. Among the other six attributes, the tracker carries this channel, but only together with its two connection handles (Figure 4b). Alone it repairs 13 of the 15 and damages a row that grading had left intact, and the two handles alone damage two, so a partial repair can itself interfere. Restoring all seven leaves all 165 evaluations as in the forked arm, requirement by requirement (Table 8).

The direction of the error. Both channels remove work the agent did from what the evaluator scores. On the 16 census rows with an interference event, grading turned 60 of their 113 passing assertions into failures and none of their 18 failing ones into passes. Grading could raise a score only if the lost work had been a mistake, such as one that breaks an assertion an agent doing nothing would pass. Eligible rows hold only 3 failed assertions of that kind among 980, none on those 16 rows. The direction is a property of the grader and its tests. τ-bench's grader writes the answer, so there extra grading manufactures successes (§3.1). A one-sided error shifts a training signal rather than widening it, and averaging over episodes does not remove a shift.

5 Repairs, Guarantees and Boundaries

Each channel we found needs a later reader of the grader's writes, so every repair we test keeps those writes from that reader. They differ in what they protect and leave open.

A forked grader. We run the grader in a disposable child process. This shifts the guarantee from listing everything the grader may change to restricting where its writes are visible. Our attempt at

the former, a fingerprint of all of the environment's in-memory state, changed on every call even on two tasks without interference (Appendix L).

Theorem 1 (Isolation by fork)**.** *Let the grader run in a child created by `fork()`, returning its result over a pipe, with its address space discarded, and suppose every shared mapping the parent holds at the moment of the fork is backed by a named file. Then a store the grader makes to its own address space reaches the parent only through the pages of such a file, and the grader can influence the parent only through the file system, inherited descriptors, or state outside both address spaces.*

Copy-on-write keeps the child's writes private only for private mappings: a region mapped `MAP_SHARED` survives the fork uncopied, and the child can make it writable. The theorem restates ordinary process semantics; what we add is a check of its condition on the running system (Appendix K). At every fork `/proc/self/maps` shows no writable shared region and at most 13 read-only ones, which a diagnostic replay finds to be SQLite's maps of the twelve base databases and a read-only system cache. An audit hook records every write-mode open, socket, subprocess and memory map the child makes. SQLite writes from C, below the hook, and each database map ends at its file's end, so a store through one changes the file. The task and database directories are therefore hashed around every call. Across 5,375 forked calls in 825 cells, no directory changed, the child created no socket, subprocess or map, and no recorded outcome moved. The descriptor channel stays open, since inherited descriptors share offsets and locks a content hash cannot see. A forked call takes a median 262 ms against 113 ms in process, over our declared $1.5\times$ budget.

Other repairs. A cached first grade repairs the τ-bench report but not the first call's write, and AppWorld's snapshot endpoints isolate the grader but leave the world unclosable (Appendix H).

Scope of the hazard. Twelve of the seventeen environments we read, from sixteen packages, cannot fail this way, most because the environment's step function returns the reward and no separate grader runs (Appendix I). On AppWorld the damage stops when a world closes: a world built after closing a graded one was unaffected in 55 of 55 families, since it sets up its applications afresh. ScienceWorld (Wang et al., 2022) can fail this way, but its runs disagree with themselves on 12 of 60 episodes, too often for a with-and-without comparison to detect an effect (Appendix J).

6 Related work

Audits of agent benchmarks ask whether a verdict is right, and they have found contamination, construction defects and wrong verdicts (Oren et al., 2024; Zhu et al., 2025; Dong et al., 2026). We ask whether a correct verdict changes the next one. Several systems already grade on copies, snapshots or read-only state (Cui et al., 2026; Lu et al., 2025; Zhou et al., 2026), and we state the condition these designs protect. Non-interference comes from security (Goguen & Meseguer, 1982), and our paired runs and forked grader follow self-composition (Barthe et al., 2004) and secure multi-execution (Devriese & Piessens, 2010). Appendix O discusses each line of work in full.

7 Discussion and limitations

Implications for benchmarks. (i) State when the grader runs and whether anything later reads its writes (§4.2). (ii) Test the whole pipeline, not the grader alone: a grader correct for its input can still change the next score (§3.1). (iii) Choose the protection by what is shared, and say what stays open. A fork protects memory, not files, descriptors or other processes. A cached grade protects the report but not the world, and a disposable page protects the agent's page but not the server.

Limitations. Replay holds actions fixed, and our AppWorld live runs are too small to estimate how often this happens with live agents (repeat floors 2 and 0 of 55). The closed-loop τ-bench contrasts give size and direction only (§3.1). The published AppWorld runs graded only at the end, and no published result we found used the analysis script, so we show no published number to be wrong. In both live WebArena runs the agent solved too few tasks to pass our competence gate, so the browser claim stops at disturbance. Six pre-declared gates, two predictions fixed in advance and part of a third failed, each reported beside the claim it limits (Table 18). These pipelines show that the hazard exists and, on AppWorld, how it arises, not how often it occurs. Measuring that rate needs a census of deployed training and evaluation pipelines, which we leave to future work. Whatever that rate, a benchmark should report its grading schedule as part of the treatment, not assume a passive read.

REPRODUCIBILITY STATEMENT

The replay censuses, mechanism interventions and source audits make no model calls. The live ACE, closed-loop τ-bench, public-wrapper and browser experiments generate new model outputs under the settings reported in their appendices. Appendices B to G identify the environment versions, interventions, populations, protocols and failed gates. The artifact includes analysis and experiment code, task-free aggregates, file hashes and executable consistency checks. It also holds the fixed action sequences that the AppWorld replays use, for the 165 census rows and the 130 held-out rows, password-protected to keep test-set content out of web crawls, with commands that rerun the sweeps, restorations, isolation checks and the held-out check without a model. Model outputs and world databases remain on the experiment host. In their place the artifact carries a fingerprint of every record file the original rollouts saved and a script that recomputes each eligibility decision from a fresh replay. On our host it reproduces 164 of the 165 decisions, and the exception is a row whose replay noise the filter had removed (Appendix B). The live-agent runs call models and are not recomputable from the artifact, and the aggregate checks do not replace rerunning them. Public task identifiers in Appendix A identify the affected AppWorld families. The two mechanisms are tied to pinned source lines and independently restored state components in §4.3. The protocol corrections preserve earlier versions and distinguish framework errors from task failures (Appendices M and N).

ETHICS STATEMENT

We study public benchmark code and synthetic benchmark tasks. The reported failures concern measurement validity; they do not establish a security exploit or incorrect historical leaderboard scores. The manuscript identifies affected code and gives tested remedies with their limits, including prior copy-scoring repairs. The published τ-bench wrapper demonstrates repeated terminal grading in actual integration code; the successor benchmark τ^2-bench avoids the identified database-writing channel. We do not report experiments on human subjects or deployed user systems.

AI USE STATEMENT

We used LLMs in several parts of this work. For the manuscript, the authors supplied the draft, and LLMs polished the text and suggested its structure. For the experiments, the authors designed the experiments and LLMs wrote the code; the authors then ran the experiments and analyzed the results. LLMs also helped search and verify the literature. The authors take full responsibility for the paper.

REFERENCES

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. τ^2-bench: Evaluating conversational agents in a dual-control environment. *arXiv preprint arXiv:2506.07982*, 2025.

Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop (CSFW)*, 2004.

BrowserGym contributors. Evaluation in WebArena differs from the official implementation. GitHub issue 304, 2025. URL `https://github.com/ServiceNow/BrowserGym/issues/304`.

Micah Carroll, Rohin Shah, Mark K. Ho, Tom Griffiths, Sanjit Seshia, Pieter Abbeel, and Anca Dragan. On the utility of learning about humans for human-AI coordination. In *Advances in Neural Information Processing Systems*, 2019.

Avyay M. Casheekar and Hariganesh Tangirala. When is an agent evaluation over? Outcome finality and cross-unit separation. *arXiv preprint arXiv:2608.14940*, 2026.

Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Why do multi-agent LLM systems fail? In *Advances in Neural Information Processing Systems*, 2025.

Maxime Chevalier-Boisvert, Bolun Dai, Mark Towers, Rodrigo Perez-Vicente, Lucas Willems, Salem Lahlou, Suman Pal, Pablo Samuel Castro, and J K Terry. Minigrid & Miniworld: Modular & customizable reinforcement learning environments for goal-oriented tasks. In *Advances in Neural Information Processing Systems, Datasets and Benchmarks Track*, 2023.

Marc-Alexandre Côté, Ákos Kádár, Xingdi Yuan, Ben Kybartas, Tavian Barnes, Emery Fine, James Moore, Ruo Yu Tao, Matthew Hausknecht, Layla El Asri, Mahmoud Adada, Wendy Tay, and Adam Trischler. TextWorld: A learning environment for text-based games. In *Computer Games Workshop at IJCAI 2018*, 2018.

Chaoqun Cui, Jing Huang, Shijing Wang, Liming Zheng, Qingchao Kong, and Zhixiong Zeng. Agentic reward modeling: Verifying GUI agent via progressive trajectory-grounded interaction. *arXiv preprint arXiv:2602.00575*, 2026.

Gaole Dai, Shiqi Jiang, Ting Cao, Yuqing Yang, Yuanchun Li, Rui Tan, Mo Li, and Lili Qiu. ProRe: A proactive reward system for GUI agents via reasoner–actor collaboration. In *International Conference on Learning Representations*, 2026.

Dominique Devriese and Frank Piessens. Noninterference through secure multi-execution. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, pp. 109–124, 2010.

Zihan Dong, Zhiyuan Ma, Zekun Wang, Yunqing Li, Zirou Liu, Ruixuan Deng, Qishi Zhan, and Rui Qian. How benchmarks mis-score computer-use agents. *arXiv preprint arXiv:2607.28367*, 2026.

Tom Everitt, Marcus Hutter, Ramana Kumar, and Victoria Krakovna. Reward tampering problems and solutions in reinforcement learning: A causal influence diagram perspective. *Synthese*, 198 (Suppl 27):6435–6467, 2021.

Lang Feng, Zhenghai Xue, Tingcong Liu, et al. Group-in-group policy optimization for LLM agent training. *arXiv preprint arXiv:2505.10978*, 2025.

Jason Gait. A probe effect in concurrent programs. *Software: Practice and Experience*, 16(3): 225–233, 1986. doi: 10.1002/spe.4380160304.

Joseph A. Goguen and José Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*, pp. 11–20, 1982.

Leon Guertler, Bobby Cheng, Simon Yu, Bo Liu, Leshem Choshen, and Cheston Tan. TextArena. *arXiv preprint arXiv:2504.11442*, 2025.

Danijar Hafner. Benchmarking the spectrum of agent capabilities. In *International Conference on Learning Representations*, 2022.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, 2024.

Jing Yu Koh, Stephen McAleer, Daniel Fried, and Ruslan Salakhutdinov. Tree search for language model agents. *arXiv preprint arXiv:2407.01476*, 2024.

Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. iDFlakies: A framework for detecting and partially classifying flaky tests. In *Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pp. 312–322, 2019.

Thibault Le Sellier de Chezelles, Maxime Gasse, Alexandre Lacoste, Massimo Caccia, Alexandre Drouin, Léo Boisvert, Megh Thakkar, Tom Marty, Rim Assouel, Sahar Omidi Shayegan, Lawrence Keunho Jang, Xing Han Lù, Ori Yoran, Dehan Kong, Frank F. Xu, Siva Reddy, Graham Neubig, Quentin Cappart, Ruslan Salakhutdinov, and Nicolas Chapados. The BrowserGym ecosystem for web agent research. *Transactions on Machine Learning Research*, 2025. URL `https://openreview.net/forum?id=5298fKGmv3`.

Evan Zheran Liu, Kelvin Guu, Panupong Pasupat, Tianlin Shi, and Percy Liang. Reinforcement learning on web interfaces using workflow-guided exploration. In *International Conference on Learning Representations*, 2018.

Jiarui Lu, Thomas Holleis, Yizhe Zhang, Bernhard Aumayer, Feng Nan, Haoping Bai, Shuang Ma, Shen Ma, Mengyu Li, Guoli Yin, Zirui Wang, and Ruoming Pang. ToolSandbox: A stateful, conversational, interactive evaluation benchmark for LLM tool use capabilities. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 1160–1183, 2025.

Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 265–276, 2009.

Joe Needham, Giles Edkins, Govind Pimpale, et al. Large language models often know when they are being evaluated. *arXiv preprint arXiv:2505.23836*, 2025.

Yonatan Oren, Nicole Meister, Niladri Chatterji, Faisal Ladhak, and Tatsunori B. Hashimoto. Proving test set contamination in black-box language models. In *International Conference on Learning Representations*, 2024.

Prime Intellect community. tau-bench environment for the prime intellect environments hub, `environments/tau_bench/tau_bench_env.py`. `https://github.com/PrimeIntellect-ai/community-environments`, 2026. Commit 76ada37e (26 February 2026); the rubric's `evaluate_tau_task` calls `calculate_reward()` on the live environment after the episode ends.

Max-Philipp Schrader. gym-sokoban: A Sokoban environment for OpenAI Gym. `https://github.com/mpSchrader/gym-sokoban`, 2018.

Chenrui Shi, Yuwei Wu, Yang Liu, Ruining Feng, Zirui Shang, Zhi Gao, Lifeng Fan, and Che Sun. Interactive reward agent and GUI-RewardBench: Official repository. GitHub repository, commit a67e756929d45a776a03aa11e589b8c077de9c97, 2026a. URL `https://github.com/Kendrick-Stein/InteractiveRewardAgent-OfficialRepo`.

Chenrui Shi, Yuwei Wu, Yang Liu, Ruining Feng, Zirui Shang, Zhi Gao, Lifeng Fan, and Che Sun. Interactive reward agent: GUI task evaluation via environment-state verification. *arXiv preprint arXiv:2607.25904*, 2026b.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. ALFWorld: Aligning text and embodied environments for interactive learning. In *International Conference on Learning Representations*, 2021.

Mark Towers et al. Gymnasium: A standard interface for reinforcement learning environments. *arXiv preprint arXiv:2407.17032*, 2024.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. AppWorld: A controllable world of apps and people for benchmarking interactive coding agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 16022–16076, 2024a. doi: 10.18653/v1/2024.acl-long.850. URL `https://aclanthology.org/2024.acl-long.850/`.

Harsh Trivedi et al. AppWorld repository README, section "evaluating the agent". `https://github.com/StonyBrookNLP/appworld`, 2024b. "You can run the above at any point. This may be useful for RL rewards or just one-off exploration".

Harsh Trivedi et al. AppWorld repository, `scripts/score_by_interaction_limits.py`. `https://github.com/StonyBrookNLP/appworld/blob/main/scripts/score_by_interaction_limits.py`, 2025. Added 15 April 2025 (commit e889247d); we run commit d47901fa, the version compatible with release 0.1.3.post1.

UK AI Safety Institute. Inspect: An open-source framework for large language model evaluations. Python package `inspect_ai` 0.3.268, 2026.

Hao Wang, Hanchen Li, Qiuyang Mang, Alvin Cheung, Koushik Sen, and Dawn Song. Do androids dream of breaking the game? Systematically auditing AI agent benchmarks with BenchJack. *arXiv preprint arXiv:2605.12673*, 2026.

Ruoyao Wang, Peter Jansen, Marc-Alexandre Côté, and Prithviraj Ammanabrolu. ScienceWorld: Is your agent smarter than a 5th grader? In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 11279–11298, 2022.

Zihan Wang, Kangrui Wang, Qineng Wang, et al. RAGEN: Understanding self-evolution in LLM agents via multi-turn reinforcement learning. *arXiv preprint arXiv:2504.20073*, 2025.

Zhiheng Xi, Jixuan Huang, Chenyang Liao, et al. AgentGym-RL: Training LLM agents for long-horizon decision making through multi-turn reinforcement learning. *arXiv preprint arXiv:2509.08755*, 2025.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*, 2024.

Huining Yuan, Zelai Xu, Huaijie Wang, Xiangmin Yi, Jiaxuan Gao, Xiao-Ping Zhang, Yu Wang, Chao Yu, and Yi Wu. Verifiable process rewards for agentic reasoning. *arXiv preprint arXiv:2605.10325*, 2026.

Yang Yue, Zhiqi Chen, Rui Lu, Andrew Zhao, Zhaokai Wang, Yang Yue, Shiji Song, and Gao Huang. Does reinforcement learning really incentivize reasoning capacity in LLMs beyond the base model? In *Advances in Neural Information Processing Systems*, 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. *arXiv preprint arXiv:2510.04618*, 2025.

Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D. Ernst, and David Notkin. Empirically revisiting the test independence assumption. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA)*, pp. 385–396, 2014.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. WebArena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=oKn9c6ytLx.

Xuhui Zhou et al. OdysSim: public Tau-USI runtime, agents/tau_usi/runtime.py. https://github.com/sunnweiwei/OdysSim, 2026. Added 9 September 2026 (pull request 23, commit 7b7df3d). Its reward endpoint scores a deep copy of the environment; accessed 25 September 2026.

Yuxuan Zhu, Tengjun Jin, Yada Pruksachatkun, Andy Zhang, Shu Liu, Sasha Cui, Sayash Kapoor, Shayne Longpre, Kevin Meng, Rebecca Weiss, Fazl Barez, Rahul Gupta, Jwala Dhamala, Jacob Merizian, Mario Giulianelli, Harry Coppock, Cozmin Ududec, Jasjeet Sekhon, Jacob Steinhardt, Antony Kellermann, Sarah Schwettmann, Matei Zaharia, Ion Stoica, Percy Liang, and Daniel Kang. Establishing best practices for building rigorous agentic benchmarks. *arXiv preprint arXiv:2507.02825*, 2025. URL https://arxiv.org/abs/2507.02825.

APPENDIX

Each appendix below answers one question the body leaves open (Table 4).

Table 4: The question each appendix answers, and where the body raises it.

	question	body
A	Which population and filter does each count use, and which families are affected?	all sections
B	How were the AppWorld replays and live agents run, and what does each trajectory set show?	§2, §3.2, §4
C	What does the public τ-bench wrapper record, and what does a replay of public trajectories show?	§3.1
D	How large is the closed-loop τ-bench effect against its repeat floor?	§3.1
E	What does grading after every action do to AppWorld's published runs?	§3.2
F	What does AppWorld's own analysis script change?	§3.2
G	Does BrowserGym's adapter move the page, and does a live agent's outcome change?	§3.3
H	Why can AppWorld's snapshot endpoints not serve as the repair?	§4.1, §5
I	Which environments can hold the hazard at all?	§5
J	Why is ScienceWorld reported as unresolved?	§5
K	Why can a forked grader reach the parent's memory only through files?	§5
L	Why did our in-memory fingerprint not certify a clean call?	§5
M	Which declared gates and predictions failed, and which claims were withdrawn?	§4.3, §7
N	Which defects in our own instruments occurred, and what did they change?	—
O	Which prior work is closest, and how does this paper differ from it?	§6

A UNITS AND POPULATIONS BEHIND EVERY COUNT

This appendix defines every count. Five unit nouns name AppWorld populations, and they are not interchangeable. A *family* is one AppWorld task family, represented in our experiments by one of its tasks, such as `325d6ec_1`. A *trajectory set* is one model's recorded output over all 55 families, and we use three. A *row* is one family under one trajectory set, so there are 165. An *episode* is one replay of a row, and a *cell* is one row under one grading schedule, run in each arm. Table 5 gives each experiment's population and the filter behind every smaller denominator. The samples are fixed selections with repeated rows within task families, so we report exact counts. A zero count is no guarantee outside the measured configurations, and we attach no confidence bounds and infer no population incidence from these correlated rows.

The families, named. AppWorld task identifiers are public, so we give them. Interference falls on seven families out of 55: `325d6ec_1`, `59fae45_1`, `6f4b9a5_1`, `83a7951_1`, `986aa4e_1`, `b6d1104_1` and `d18139b_1`. The black-box HTTP setup moves the official evaluation on all seven and on none of the other 47 that completed every arm, and destroys a success on three of them (§4.1). Under the census, grading destroys successes on four of the seven: `325d6ec_1`, `59fae45_1`, `6f4b9a5_1` and `986aa4e_1`. Of the two channels of §4.3, `986aa4e_1` alone belongs to the counter-key channel: its saved records are byte-identical between arms while its grade changes, no application data file differs under HTTP (the counter file, `model_hashes.json`, differs on all 54 families), and restoring `home_path` repairs it. The other six belong to the tracker channel: their saved records differ, an application's data file differs under HTTP (`spotify` for `325d6ec_1` and `59fae45_1`, `simple_note` for `6f4b9a5_1` and `b6d1104_1`, `venmo` for `83a7951_1` and `d18139b_1`), and restoring the other six attributes repairs them.

Six in the census, seven over HTTP. Six of the seven carry their events on rows the census counts. The seventh, `d18139b_1`, carries two, under the Qwen3-32B and Qwen3.5-27B trajectory sets, on rows the census excludes because the replay did not reproduce the recorded trajectory exactly. Its third row, under Llama-70B, is kept and shows no interference. Interference appears on 2 of the 14 excluded rows and 16 of the 151 kept, which does not prove the exclusions unbiased. The excluded rows carry no destroyed success and no change in the forked arm. Both populations together give 18 of 165 (10.9%, against 10.6% on the kept rows), which is also the cadence sweep's count at its highest cadence.

Table 5: Each experiment, its unit, its denominator and filter, and its floor.

experiment	unit	denominator and filter	floor
in-process replay census (§4.3)	row	165 run; 151 eligible after excluding 14 whose replay did not reproduce the recorded trajectory exactly	0 of 165
cadence sweep (§4.2)	cell	165 per cadence, 5 cadences, no cell failed	forked arm 0 of 825
placement sweep (§4.2)	cell	165 per placement, 5 placements, one grading call in every cell	forked arm, same cells
mechanism intervention (§4.3)	row	165, graded after every action, in-process and forked arms plus seven restoration arms	forked arm is the reference
held-out restoration check (App. M)	row	139 run; 130 eligible `test_challenge` rows, five arms, prediction fixed before the run	forked arm is the reference
live attribution (App. B)	sequence	3 fixed action sequences of the two cleared live families, six arms	repeated forked arm
black-box HTTP (§4.1)	family	55, four arms against the shipped server, 54 completed every arm	repeated control arm, 0 of 54
live ACE agent, Qwen3-32B (§4.1)	family	55, two arms plus a repeat of one	2 of 55
live ACE agent, Llama-3.3-70B (§4.1)	family	55, two arms plus a repeat of one; grouping from its own census, 4 flagged	0 of 55
training loop (§3.2)	episode	165 run; 94 is the denominator for a destroyed reward, being the episodes that completed the task	forked arm is the reference
cross-boundary (§5)	family	55, a second world built after closing a graded one	—
WebArena (§3.3)	intent template	50, fixed before any outcome was read	—
ScienceWorld (§5)	task × variation	60, from 30 tasks and 2 variations	12 of 60
τ-bench replay (App. C)	trajectory	1,980 public; 153 replay exactly, 633 after one rename, 1,318 rejected by the benchmark's own parser	repeated replay agrees on all 662 the parser accepts
τ-bench wrapper, end to end (§3.1)	episode	165 run; 159 with the benchmark's own grade, 6 step-capped excluded	paired within the episode
τ-bench repaired rubric (§3.1)	episode	165 new episodes; 161 ended, 3 step-capped graded on a copy, 1 model-service error	—
τ-bench closed loop (§3.1)	task	165 per arm, four arms, four model setups	repeats differ on 31 and 36; 24 and 33; 17 and 7; 35 and 4
τ-bench second grading call (App. C)	episode	50 public gpt-4o episodes scored 0, first 25 per domain that end under replay; 49 fail under replay	deterministic replay
AppWorld's analysis script (§3.2)	row	165, three identical experiments per set, 61 grading calls per task	identical builds agree on 165 of 165
BrowserGym exposure (§3.3)	task	812 in WebArena's task file, 302 with a navigating target under the upstream evaluator	source count, no arms
BrowserGym live (§3.3)	task	66 `shopping_admin` tasks with a fixed-URL target, under the packaged and the upstream evaluator; 64 with both primary arms under the upstream one	repeats of both arms on 15
AppWorld published runs (§3.2)	rebuilt task	672 published; 666 rebuilt, 662 reproduce, 526 eligible	eligible: repeat and forked 0 of 526; all 666: forked 2, repeat 2

B SETTINGS AND RESULTS OF THE APPWORLD EXPERIMENTS

This appendix gives the settings shared by the AppWorld replays and live agents of §2 and §4, the second live-agent model, and the results by trajectory set.

B.1 SETTINGS SHARED BY THE APPWORLD EXPERIMENTS

Version and configuration. Every worker aborts unless the AppWorld distribution is `0.1.3.post1`, and the interpreter is CPython 3.11.14. The three trajectory sets, A, B and C, were recorded by Llama-3.3-70B-Instruct, Qwen3.5-27B and Qwen3-32B, one trajectory per family. Each is a rollout of the ACE agent, the checked generator with its published playbook, which carries no ground truth, under greedy decoding with a 131,072-token context and a 2,048-token output cap, in a fresh world with seed 123 and at most 40 interactions. Replays build their worlds with the configuration recorded in the trajectory. A family's representative is the lexicographically first of its public `test_normal` tasks, fixed by a zero-model-call screen before any census. Every count excludes the family in which the effect was first noticed, so the census uses 55 of the 56. Ground truth enters the agent's world only in experiments with an arm that calls the public `evaluate` method, which refuses to run without it, and there every compared arm loads it.

Process discipline. Each arm of each family runs in its own process, which exits when the arm ends. Every reported outcome comes from the offline evaluator in a fresh subprocess after `close()`, the path that produces leaderboard numbers, so corrupted in-memory state can reach a reported number only through what was persisted.

Experiment naming. Output directories are keyed by trajectory set, arm, task and shard, and reruns use fresh names; Appendix N records the two collisions that made each part of that key necessary.

Controls on the fork checks. Three controls test our memory-map check: a child's store through a private mapping leaves the parent's copy intact, a store through a shared mapping reaches the parent, and the parser finds a shared mapping we deliberately hold. The grader's databases are shared-cache in-memory SQLite, so no write-ahead-log index sits on disk to be mapped. Each forked child exercises both the builtin `open` and `os.open` and checks that its audit hook saw both. The directory hash is checked against perturbations of a live directory, including `model_hashes.json` itself, and in a side run with the grader's report switched on it caught exactly the two report files the grader writes.

Eligibility. A row is eligible when the ungraded and forked arms both replay the recorded trajectory exactly, reproduce the original run's records and agree with the historical outcome for that task. The in-process arm never enters the decision, so eligibility cannot select on the effect being measured. On the defect-free census 14 of 165 rows are ineligible, all because a replay did not reproduce the recorded outputs exactly. In 13 both arms diverge from the recorded trajectory. In `29a7b7e_1` only the ungraded arm did, and a later replay reproduces it in both. This is replay nondeterminism rather than an effect of grading, and the filter removes it from both sides. The artifact's eligibility check replays each row against fingerprints of the original records and reproduces the census's decision on the other 164 rows.

Compute. The AppWorld replays, sweeps and interventions are CPU-only and make no model calls. The live-agent arm serves Qwen3-32B on one GPU with greedy decoding and drives it with the ACE agent at commit `928e8687` and its pinned playbook, both checked by hash before each rollout. Appendices C, D and G give the models and settings of the other experiments that call models.

A second model in the live arm. The Llama-3.3-70B-Instruct run of §4.1 follows a preregistered protocol (`live_ace_llama_prereg_20260925.md`; analysis `live_ace_llama_matched.py`) on the same 55 families in the same order. One amendment, made before any episode completed, serves FP8 weights on one GPU through a copy of the audited server that declares and verifies the quantization, with a 131,072-token context; decoding, agent, playbook and checks are unchanged. The protocol predicted that the flagged families would move, that the cleared ones would not move beyond the floor, and that no success would be manufactured. The first and third held, and the second failed on 2 of the 51 cleared families (§4.1). All 55 families completed both arms and the repeat without framework or grading errors, the repeat agrees with the isolated

arm on all 55, and the graded arm passes 17 families where the isolated one passes 19. A zero-call diagnostic finds that the agent's actions diverged between the arms in two families, one flagged and one of the two cleared ones.

The two cleared families that moved. Both are tracker-channel families that the census flags on the Qwen trajectory sets (Appendix A). Before replaying their live actions we fixed a prediction in advance (`LIVE_ATTRIBUTION_PREREG_20260926.md`, SHA-256 `b73ab8b6...`) and ran the restoration worker of §4.3 at the live cadence, with a repeated forked arm. In `59fae45_1` both arms executed the same ten actions. On them in-process grading destroys the success, restoring `home_path` does not repair it, restoring the other six or all seven does, and the repeated forked arm agrees. That prediction held: this is the tracker channel on a path the census trajectory did not take. In `d18139b_1` the arms first differ at action 24, and in-process grading changes the evaluation on neither arm's own sequence, so the second prediction failed. A later diagnostic replays the 23 shared actions. Under forked and in-process grading the outputs first differ at action 23, a request approving this month's Venmo payments. Forked grading reports them already completed, and in-process grading reports an insufficient balance, as in the two live arms. Restoring the two connection handles after every call, alone or with the rest, keeps the forked output, and restoring `home_path` or the tracker alone does not. The grader's overwriting of the handles changed what the action returned, and the agent's different next actions changed its outcome.

The five reward vectors. Figure 5 draws the five episodes §3.2 describes, aligned on the step at which each task is completed.

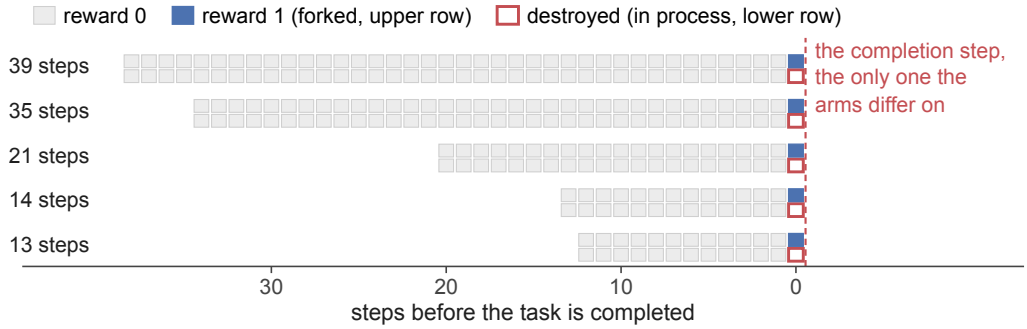

Figure 5: **The reward vectors agree on every step except the one where the task is completed.** The five episodes whose vectors differ, aligned on the completion step rather than on the first, since their lengths range from 13 to 39 steps. In each, the forked arm returns a reward at the final step and the in-process arm returns none.

Two readouts of the same loop. A training loop reads the in-process reward, and a leaderboard reads the saved world. Figure 6 joins the two for the same 165 trajectories, each against its forked arm. Six trajectories lose a success in only one readout. In the five that lose only the saved-world success, the last in-process grade still reports success.

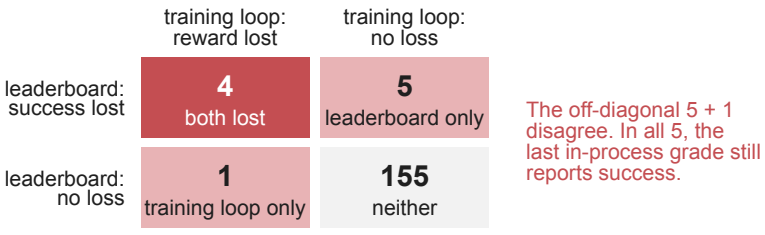

Figure 6: **A training loop and a leaderboard can disagree about one episode.** Our 165 trajectories, graded in process after every step and compared with their forked arms. Rows give the saved world's official evaluation, columns the in-process reward at completion, which only the 94 episodes that complete the task can lose.

B.2 Results by trajectory set, placement and restoration

Table 6 breaks the cadence curve out by the trajectory set that produced the replayed actions. No set changes at a single terminal call, and none decreases as the cadence rises, so the curve is not an artifact of one model's trajectories.

Table 6: Families whose recorded outcome changed, and successes destroyed, by trajectory set. Set rows count 55 families each. Set A is Llama-3.3-70B-Instruct, B is Qwen3.5-27B and C is Qwen3-32B.

	end	1 in 8	1 in 4	1 in 2	every step
set A: changed / destroyed	0 / 0	1 / 1	2 / 2	3 / 2	4 / 2
set B: changed / destroyed	0 / 0	3 / 1	4 / 2	6 / 3	7 / 4
set C: changed / destroyed	0 / 0	4 / 1	4 / 1	4 / 1	7 / 3
mean grading calls per episode	1.0	1.8	4.0	8.4	17.3

Table 7 gives the single-call sweep of §4.2 by placement, as it was designed and run; Figure 3b regroups its cells by the actions that follow the call.

Table 7: The single-call sweep by placement. Each cell makes exactly one grading call, after action $\max(1, \lceil fn \rceil)$ of n, and each count compares the in-process arm with a matched forked arm on the same cells. The same 825 cells, regrouped by the actions left after the call, give Figure 3b.

placement f	0	$\frac{1}{4}$	$\frac{1}{2}$	$\frac{3}{4}$	1
rows	165	165	165	165	165
outcome changed	3	3	4	7	0
successes destroyed	3	3	3	4	0
successes manufactured	0	0	0	0	0

Table 8 gives every restoration against all 165 rows, not only the 18 that grading damages. Only restoring all seven leaves every evaluation as in the forked arm, and two partial restorations change rows that grading had left intact.

Table 8: Every restoration against all 165 census rows, grading after every action. Damaged rows are grouped as in Figure 4b. A row is repaired, or unchanged, when its official evaluation equals the forked arm's, requirement by requirement. No arm manufactures a success.

	damaged rows repaired		other rows	successes
restored after every call	records unchanged	records changed	changed	destroyed
nothing (in-process grading)	0 of 3	0 of 15	0 of 147	9
`home_path`	3 of 3	0 of 15	0 of 147	6
the two connection handles	0 of 3	0 of 15	2 of 147	10
the three file-name attributes	0 of 3	0 of 15	0 of 147	9
the tracker alone	0 of 3	13 of 15	1 of 147	5
the tracker and the two handles	0 of 3	15 of 15	0 of 147	3
the other six	0 of 3	15 of 15	0 of 147	3
all seven	3 of 3	15 of 15	0 of 147	0

Figure 7 compares four ways to detect an interference event, and Table 9 compares the census under three instruments. The defect that prompted the re-census moved no reported number.

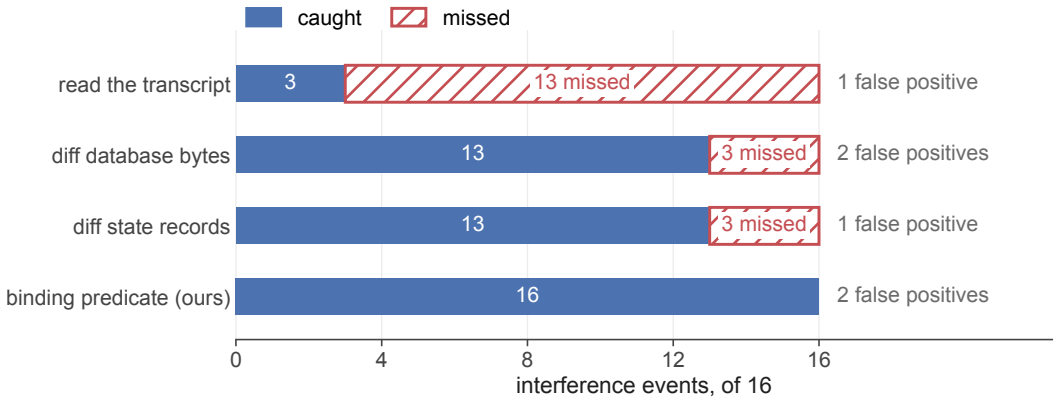

Figure 7: **The counter file is not a saved record, so no comparison of records catches every event.** Sixteen interference events on 151 eligible rows. Reading the transcript misses 13 of them, diffing the database bytes or the state records misses 3, and a predicate that watches the grader's class bindings misses none. False positives are annotated rather than stacked, since a false positive costs an inspection while a miss costs a wrong number.

Table 9: The census under three instruments. The columns are separate runs, not nested subsets, so the middle one can carry more events on fewer rows. *original* is the September 18 run, whose read of the counter file materialized and persisted entries (Appendix N). *re-run, sound rows* fixed that read but acquired a shard collision, so it keeps only the 146 rows whose arms were provably matched; it is the only run in which the binding predicate missed an event. *defect-free* is the September 23 re-census, the source of every census number in the body.

	original	re-run, sound rows	defect-free
rows / eligible	165 / 151	146 / 134	165 / 151
interference events (families)	16 (6)	17 (7)	16 (6)
successes destroyed, none manufactured	9	9	9
records identical, grade changed (families)	3 (1)	4 (2)	3 (1)
forked arm disagrees with ungraded	0 of 165	0 of 146	0 of 165
repeated ungraded arm disagrees	not run	not run	0 of 165
binding predicate missed	0	1	0

C THE τ-BENCH MEASUREMENT

Appendix C.1 gives the public wrapper run of §3.1 and its repair, and Appendix C.2 an exploratory replay census of the public trajectories, whose sample-size gate failed.

Pinned versions. τ-bench at commit `59a200c6`, with `envs/base.py` at SHA-256 `1ad04020...`. There `calculate_reward` reassigns `self.data` to freshly loaded initial data (line 133) and calls `self.step` on every ground-truth action (line 136), each of which is appended to the environment's action log.

C.1 THE PUBLIC WRAPPER THAT GRADES TWICE

A pipeline that grades twice. Nothing in τ-bench's documentation invites an early call. Our search of public RL frameworks and wrappers (verl, verl-agent, SkyRL, rLLM, AgentGym-RL, RAGEN, agent-lightning, OpenRLHF, gem, EnvScaler, DoomArena, ART and others) found none that calls `calculate_reward` at a non-terminal step, and one that calls it after the terminal step. That one, the τ-bench environment on the Prime Intellect hub (Prime Intellect community, 2026), steps the live environment until done, when the benchmark's `step` has computed the reward and left the ground-truth state in place, and its rubric then calls `calculate_reward` on the same object (`environments/tau_bench/tau_bench_env.py`, `evaluate_tau_task`). That call

hashes the ground truth against itself, so it can fail only on the output check. We replayed this call shape (`taubench_double_grade.py`) on the first 25 public gpt-4o trajectories per domain that the benchmark scored 0 and that end under replay. Of these 50, 49 still fail, and the second call scores 40 of the 49 as passes: all 40 whose task has no output check, and none of the 9 whose task has one. Replay holds actions fixed, so it cannot show what an agent would do with that state. One other public wrapper, OdysSim (Zhou et al., 2026), deep-copies the environment before scoring and says why: "Upstream calculates ground truth by mutating the env. Score a copy so repeated reward requests cannot turn a failed task into a success."

The pipeline run end to end. The run of §3.1, with Llama-3.3-70B-Instruct as policy and user simulator, uses the environment's own evaluation command (`vf-eval`, verifiers 0.3.1) and its 30-step cap. The protocol stated temperature 0 for the user as well and records the correction; neither the paired comparison within an episode nor the repair depends on the user's sampling. Besides the recorder, a guard ends a rollout whose user simulator raises instead of aborting the batch, and it never fired in this batch (`tau_bench_env_probe.py`; protocol `taubench_pipeline_e2e_prereg_20260925.md`, fixed after a three-task dry run).

Result. Table 10 gives the counts behind §3.1. The six step-capped episodes are excluded because the rubric's call is then the first. On the 5 excess passes whose task has an output check, that check passed on both calls and only the database check moved. In the zero-call diagnostic, every task without an output check passes. A rubric returning the recorded terminal reward would agree with the benchmark on all 159 by construction, so the 105 is the whole effect of the second call.

The repair, run in the same pipeline. The repair is Amendment 1 of the protocol (`tau_bench_env_fixed.py`). The rubric returns the grade the benchmark's `step` computed when the episode ended, and when the step cap left no grade it makes the only grading call on a deep copy. The three step-capped episodes each scored 0. The episode with a model-service error, in airline, stopped on its first turn; its rubric scored a copy and returned 1, with no grade from the benchmark to compare. The 164 error-free rollouts contain 25 passes, and including that episode would give 26.

Checks and corrections. The first aggregator labeled every missing terminal grade a step cap; the corrected zero-call analysis separates the error by its recorded stop condition and keeps the earlier aggregate. The original data hash did not change across the 165 rubric calls. That check starts at rubric entry, after the benchmark's own grading. Caching keeps a second grade from reading the ground-truth state but does not undo the first call's write. Two earlier *unrepaired* retail batches aborted on a user-simulator exception. Their partial results are retained but not pooled with the complete unrepaired batch of Table 10, which had no rollout errors.

Table 10: The Prime Intellect τ-bench environment run end to end, as shipped and with the repaired rubric. The benchmark's terminal reward is what its own `step` computed; the shipped pipeline's is the rubric's second `calculate_reward` call.

	airline	retail	both
rollouts	50	115	165
ended by the step cap, no terminal reward, excluded	3	3	6
compared	47	112	159
passes by the benchmark's terminal reward	12	10	22
passes reported by the pipeline	43	84	127
benchmark failures the pipeline reports as passes	31	74	105
on tasks without an output check	31	69	100
on tasks with an output check the agent satisfied	0	5	5
benchmark passes the pipeline reports as failures	0	0	0
The repaired rubric, same pipeline, regenerated trajectories			
rollouts	50	115	165
ended, graded from the cached first grade	48	113	161
of which the rubric disagrees with the benchmark	0	0	0
step-capped, graded once on a copy (passes)	1 (0)	2 (0)	3 (0)
model-service-error rollouts, reported separately	1	0	1
error-free rollouts (passes)	49 (14)	115 (11)	164 (25)
rubric calls that changed the original data hash	0	0	0

C.2 A REPLAY CENSUS OF THE PUBLIC TRAJECTORIES, EXPLORATORY

Population and replay, fixed before any outcome was read. All 1,980 public trajectories in `historical_trajectories/`: gpt-4o on airline (200) and retail (460), and claude-3.5-sonnet-new on airline (400) and retail (920). The benchmark's own `message_to_action` rebuilds the actions, recorded user turns replace the user simulator, and the model client is stubbed so that no call is possible. Every arm of every trajectory runs in its own process. The arms are the four of §2, with grading after every step, plus one call at each of the five placements of §4.2, each paired with the same call in a forked child. The every-step loop also grades once after the step that ends the episode, after its reward is recorded, so that call can change no outcome. The protocol's "after every non-final step" described the calls that can, and Table 11 counts the final one too.

Eligibility, and one amendment. A trajectory is eligible when its ungraded replay returns the recorded reward, ends at the recorded step, and reproduces every recorded tool output character for character. A four-trajectory smoke test, read before any graded arm, found that the public user data renamed the field `province` to `state` after recording. We kept the character rule as primary and fixed one secondary comparison that applies that rename to the recording and nothing else.

Gates, written down first. (1) A repeated ungraded replay agrees with the first on every trajectory. (2) Every forked arm returns the ungraded reward and observations. (3) At least 1,000 trajectories are eligible. (4) Paired arms grade at the same steps. Gates (1), (2) and (4) pass on every trajectory that runs. Gate (3) fails, so we read the census as exploratory. All 1,318 replay failures have one cause: the claude recordings store each tool call's arguments as a parsed object, which `message_to_action` rejects. The character rule admits 153 trajectories and the rename comparison 633.

Table 11: τ-bench, eligible trajectories, in-process grading against the ungraded replay. Every forked arm changes nothing.

	character rule	after the rename
eligible (successes / ended failures / never ended)	153 (82 / 69 / 2)	633 (336 / 290 / 7)
every step: successes manufactured	59	238
every step: successes destroyed	0	0
one call at $f = 0, 1/4, 1/2, 3/4$: manufactured	43, 43, 45, 49	144, 144, 147, 168
one call at $f = 0, 1/4, 1/2, 3/4$: destroyed	2, 2, 0, 0	17, 17, 13, 10
one call after the last action	0	0
observations changed by grading	105 of 1,474	1,354 of 8,557
per-step rewards manufactured / lost	566 / 2	4,598 / 22

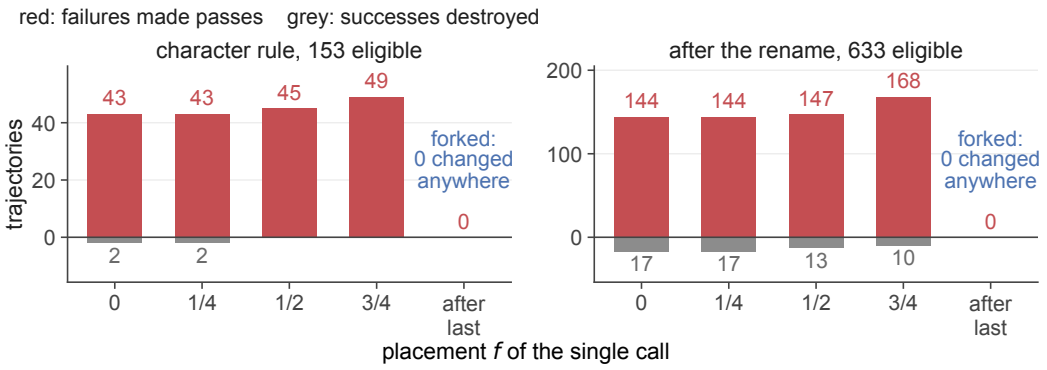

Figure 8: **On τ-bench an early grading call manufactures passes, and a call after the last action changes nothing.** One in-process call per eligible public trajectory, after action $\max(1, \lceil fn \rceil)$ of n. Red counts ungraded failures turned into passes and grey counts successes destroyed; the forked call changes no outcome at any placement.

Reading the table. Figure 8 draws the single-call rows of Table 11. Grading after every step never destroys a success, because the last call before the final step leaves the ground-truth state in place.

A single early call can, because the agent's later writes repeat changes the ground-truth state already holds. No call falls between the last action and the official reward, which the step that ends the episode computes.

A post-hoc extension. After reading the primary result we replayed the 1,318 rejected recordings, serializing each argument object before the benchmark's own parser reads it. This is Amendment 2 of the protocol, outside the pre-registered analysis, and it counts toward no gate. All 1,320 claude recordings then replay, and the repeated replay again agrees with the first on every one. Of the 144 that replay exactly, grading after every step turns 41 of 48 failures into successes, and after the rename 403 of 487 failures among the 1,232 eligible. No success is destroyed in either count, and every forked arm changes nothing.

D THE CLOSED-LOOP τ-BENCH RUN

§3.1 and Figure 2a report that grading in process after every step adds between 29 and 79 passes on 165 tasks across four configurations. This appendix gives the setup, its amendments and every arm (Table 12).

Setup, fixed before any episode. τ-bench at commit `59a200c6`, test split, one episode per task and arm, and at most 30 steps as in the benchmark's runner. The agent is the benchmark's tool-calling loop plus a grading call after every step that does not end the episode. Agent and simulated user are both Qwen3-14B with FP8 weights, served by vLLM on one GPU with thinking off, and the agent decodes greedily. The user simulator's call carried no sampling parameters, so the server applied the model's own defaults, temperature 0.6 with top-p 0.95 and top-k 20, which we noticed only after the run. The protocol had said temperature 0 for both, and its Amendment 2 records the discrepancy. Every arm shares this user, which departs from the leaderboard's gpt-4o user. The arms are `forked`, in process (`graded`), and a repeat of each; the in-process repeat was added after a two-task smoke test in which two identical forked runs disagreed.

Results. Table 12 gives each arm's passes. For each configuration the text gives the tasks that pass in only one arm, and the floors, the tasks on which two runs of one arm disagree. In the pre-registered run, 83 tasks pass only in process and 8 only forked, and 93 and 5 in the repeats. By domain the pass count goes from 33 to 79 of 115 in retail and from 9 to 38 of 50 in airline. The floors are 31 tasks forked and 36 in process. The user simulator samples, and other runtime nondeterminism is not measured separately. With every call at temperature 0 (Amendment 2), the floors fall only to 24 and 33, so temperature 0 does not make this setup deterministic. There 82 tasks pass only in process and 3 only forked. Amendment 3 repeats the four arms with DeepSeek-flash and with GLM-5.3-flash as agent and user, through their public APIs, every call at temperature 0 and GLM's thinking off. A stronger agent leaves less room, and the effect stays. With DeepSeek-flash 36 tasks pass only in process and 7 only forked, with floors of 17 and 7. With GLM-5.3-flash the counts are 39 and 8, with floors of 35 and 4.

Table 12: τ-bench in closed loop, 165 tasks per arm. The first row is the pre-registered Qwen3-14B run; the next three are its Amendment 2 and 3 re-runs. The remaining rows describe the first run. A per-step reward is a grade the loop computed after a step; "solved" is a grade of 1.

	forked	forked, repeat	in process	in process, repeat
passed, Qwen3-14B, user at default T	42	31	117	119
passed, Qwen3-14B, every call at $T = 0$	39	39	118	117
passed, DeepSeek-flash	122	127	151	150
passed, GLM-5.3-flash	122	113	153	151
episodes that ran out of steps	21	21	23	24
tool calls answered with an error	244	278	348	303
per-step rewards saying solved (%)	11	9	69	68

E GRADING APPWORLD'S PUBLISHED BASELINE RUNS

This appendix gives the rebuild behind the published-run result of §3.2, its gates and prediction, and the per-model counts.

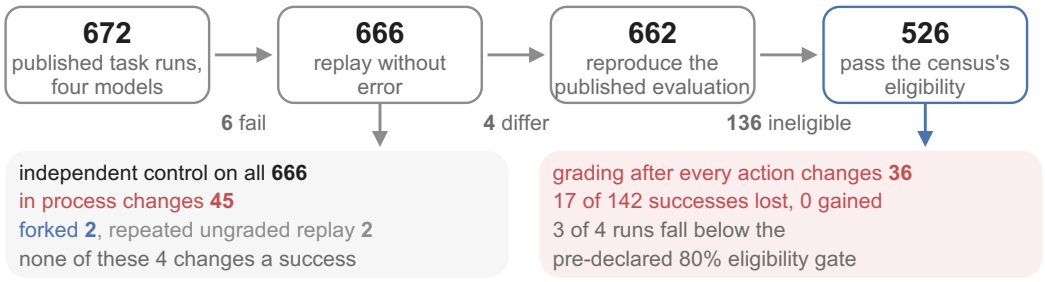

Figure 9: **The denominators of the published-run study.** The independent control branches from the 666 rebuilt tasks, because eligibility already requires the forked arm to match. Three of the four runs miss the pre-declared eligibility gate.

Population. AppWorld ships the experiment outputs of its paper's baselines (`appworld download experiment-outputs`, bundle of 27 July 2024): 28 runs, fourteen design–model pairs on each of `test_normal` and `test_challenge`. Each task carries the agent's API requests (`logs/api_calls.jsonl`: method, path and parameters, with access tokens redacted), the final databases and the published per-task evaluation. The primary population is the four ReAct runs on `test_normal`, GPT-4o, GPT-4-turbo, Llama-3-70B and DeepSeek-Coder, 168 tasks each, whose aggregates are the numbers the paper reports.

Rebuild. Each request becomes one `apis.<app>.<api>(...)` call (`official_replay.py`). Documentation reads are dropped, and access tokens come from the replayed login of the same app. Replaying the calls in a fresh world with no grading, with every output recorded, yields a trajectory our harnesses read. An original interaction was a code cell that could issue several requests, and the bundle omits the cells, so grading after every action and the script's five-action blocks run at a finer cadence than the original agent's interactions. A rebuilt task reproduces the published evaluation when it has the same success flag and the same sets of passed and failed requirements. The protocol, its gates and its prediction were fixed before any graded arm ran (`official_replay_prereg_20260925.md`), after a six-task smoke test of the rebuild alone.

Arms. The census harness of §2 runs unchanged on the rebuilt trajectories, with its own eligibility. The ungraded repeat and the forked arm must reproduce the recorded outputs exactly, the saved records must equal the published databases as a canonical multiset, and the forked evaluation must equal the ungraded and the published one. The script harness of Appendix F runs unchanged on three identical rebuilt experiments per run.

Results. Table 13 gives the counts of §3.2 by model. In the independent control over all 666 rebuilt tasks, both tasks the forked arm moves are ineligible. On the 140 rebuilt tasks outside the 526, in-process grading changes 9 evaluations and destroys 5 of their 33 ungraded successes, again manufacturing none. Of the 140, 138 fail eligibility because their saved records do not match the published databases as a canonical multiset, and 2 because the forked arm moved. The shipped script makes 10,065 to 10,248 grading calls per run, its forked copy rewrites no evaluation, and the three identical builds agree on every task.

Gates and prediction. Gate G1 asked for at least 80% of a run's tasks to be eligible. Only the DeepSeek-Coder run meets it (140 of 168, 83%); the GPT-4o, GPT-4-turbo and Llama-3-70B runs have only 125, 130 and 131 of 168 eligible (74%, 77% and 78%), which the protocol labels partially rebuilt. The rebuild reproduces the published evaluation on 662 of 672 tasks; the shortfall is the census's stricter gate, which also demands byte-exact replayed outputs and records equal to the published databases. Gates G2 and G3, that the repeated ungraded arm and every forked arm change no eligible evaluation, hold. The pre-registered prediction was that in-process grading would change some evaluations, destroy successes and manufacture none, that the script would rewrite some, all

downward, and that forked arms would change none. It held where the records can check it: none of the script's 16 rewrites turns a failure into a success, but the records do not say whether its other 8 all lowered a requirement.

Table 13: AppWorld's published ReAct runs on `test_normal`, rebuilt from their API logs and graded after every action. Successes are the published ones. From *eligible tasks* down to the rule, rows count eligible tasks; the last two count all rebuilt tasks, where only the pooled forked and repeat counts were recorded.

	GPT-4o	GPT-4-turbo	Llama-3-70B	DeepSeek-Coder	all four
published tasks / successes	168 / 82	168 / 45	168 / 35	168 / 12	672 / 174
replayed without error	166	168	165	167	666
reproduces published evaluation	165	167	163	167	662
eligible tasks / successes	125 / 59	130 / 42	131 / 30	140 / 11	526 / 142
in-process grading changes	14	11	8	3	36
of which successes destroyed	7	5	5	0	17
successes manufactured	0	0	0	0	0
repeated ungraded arm changes	0	0	0	0	0
forked arm changes	0	0	0	0	0
the shipped script changes	6	4	4	2	16
of which successes destroyed	4	2	2	0	8
its forked copy changes	0	0	0	0	0
all rebuilt: in process changes	17	14	10	4	45
forked / ungraded repeat changes					2 / 2

Scope of the result. The published runs graded once at the end, and their published numbers are not wrong. Ours say what the paper's own actions would have scored under the pipeline the benchmark's documentation invites, grading at every step, on the published population rather than on trajectories we generated. The rebuild replays requests rather than the agent's code, so the world receives the same writes in the same order but not the same Python.

F AppWorld's score-by-interaction-limit script

This appendix gives the analysis script of §3.2, its protocol and its counts on our 165 trajectories.

The shipped script. `scripts/score_by_interaction_limits.py` entered the AppWorld repository on 15 April 2025 and is on its main branch today (Trivedi et al., 2025). For each task of an experiment it parses the recorded interactions, builds a world under the experiment's own name, replays the interactions five at a time and calls `world.evaluate()` on that world after every block. That is 61 calls per task, of which the 13 within the plotted range are drawn. Every `execute` saves the replayed world into the directory it read. Building that world is itself destructive: in a smoke run whose stub crashed at the first grading call, the task's saved outputs could no longer be scored. The protocol named the main-branch version, but its import of a module absent from the pinned release stopped the smoke run, so before any graded result was read we substituted the release-compatible commit (`d47901fa`, SHA-256 `c2430599...`). The current main branch moves one import and numbers the blocks differently, and keeps the same loop.

Protocol, fixed before any run. For each trajectory set we build three identical experiments, a script copy, a floor copy and a fork copy, by replaying each recorded trajectory in a fresh world with AppWorld's default configuration and no grading. All three are scored with the offline evaluator in a fresh process. The script's source, unchanged and checked by hash, then runs on the script copy, with its plotting library replaced by a stub that records what it would draw, and on the fork copy with each `world.evaluate()` moved into a forked child. Both copies are scored again, and the floor copy is the floor.

Results. Table 14 gives each trajectory set, and Figure 10 the curves. The forked copy's last evaluation of each task matches the experiment's official one on all 165, so the copy is a faithful reference.

Table 14: AppWorld's score-by-interaction-limit script, run on experiments rebuilt from each trajectory set (55 tasks each; A is Llama-3.3-70B-Instruct, B is Qwen3.5-27B, C is Qwen3-32B).

	set A	set B	set C	all
official successes before the script runs	25	46	23	94
official evaluations the script changes	1	5	3	9
of which successes turned into failures	1	2	1	4
official evaluations the forked copy changes	0	0	0	0
identical builds that disagree	0	0	0	0
plotted points below the forked copy's (715 per set, 2,145 in all)	6	8	5	19
plotted score at 60 interactions, script / copy (%, rounded)	44 / 45	82 / 84	40 / 42	

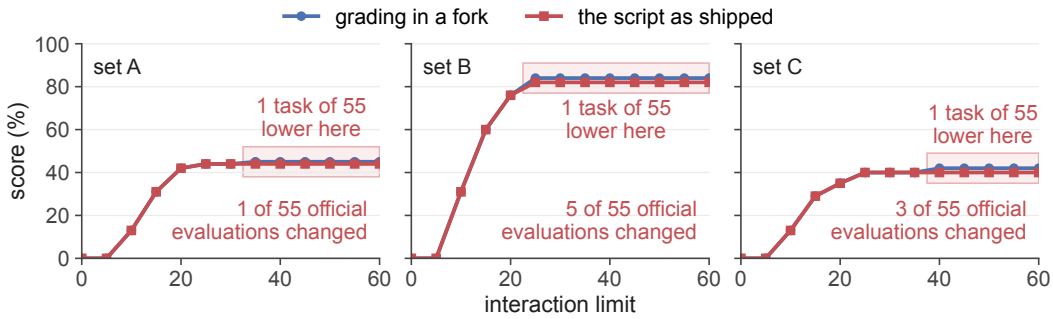

Figure 10: **The score the shipped script plots.** Red is the script as shipped, grading the world it keeps replaying; blue is the same source with each grading call in a fork. Boxes mark the limits at which the curves differ. There the script counts one task of 55 fewer, since a single task's plotted curve differs in each set. The count at lower right is how many official evaluations the script rewrites (Table 14).

G THE WEBARENA MEASUREMENT

This appendix gives the census on local test pages behind §3.3, whose nine gates all pass, and the live follow-up, whose competence gate failed in both runs.

Pinned versions. BrowserGym at commit `9e779f08`, with `env.py` at SHA-256 `867cc6ef...`; WebArena at commit `c6475f0e`, with `evaluators.py` at `6cc7b68d...` and `helper_functions.py` at `ec758da4...`; and the public `test.raw.json` at `fb1e4273...`. The runner executes the pinned `BrowserEnv._task_validate`, `HTMLContentEvaluator`, `StringEvaluator` and `gitlab_get_project_memeber_role` bodies. The only adaptations remove runtime type decorators, substitute the remote host, and replace two model-based string matchers and a tokenizer with constants. The browser is Chromium 140.0.7339.16 under Playwright 1.55.0, and no model is called.

The installed evaluator. `browsergym-webarena` does not install the upstream repository. Every release from 0.8.1 (October 2024) to 0.14.3 (January 2026) pins `libwebarena`, an unofficial port whose releases 0.0.2 to 0.0.5 (March 2024 to March 2026) all ship an `HTMLContentEvaluator` that opens a disposable page, navigates it to the target and closes it (`evaluators.py` SHA-256 `eaa0532b...`). The upstream evaluator at `c6475f0e`, and on the main branch as of 25 September 2026 (SHA-256 `e23edb39...`), navigates the page it is handed. The census below therefore describes the adapter with the upstream evaluator, and an installation from PyPI is not exposed on this channel. We learned this from the live follow-up, after the census was fixed and run; the port's repair is silent, since its package metadata carries no changelog.

Population, fixed before any outcome was read. Every distinct `intent_template_id` with at least one `program_html` target whose URL is static and is neither `last` nor `func:`, taking for each template the first eligible target in public corpus order. Selection reads configuration metadata

only, never a score, an observation or a task outcome. This yields 50 templates over five sites (GitLab, Reddit, Shopping, Shopping Admin, Wikipedia), with 17 full-page, 31 element (DOM) and 2 official-helper locators.

Arms. Fresh browser contexts under one Chromium process. The baseline opens `/agent` and records the next observation without grading. The shared arm, which grades as shipped, opens `/agent`, runs the selected target through BrowserGym's own validation method on that active page, then records the next observation. The isolated arm runs the identical target on a second disposable page in the same context, closes it, and records the original page. An observation is the triple (URL path, title, SHA-256 of the full content).

Gates, written down first. The claim passes only if all nine hold. (1) The selection contains exactly 50 templates. (2) It spans at least five sites. (3) All 50 complete in both evaluator arms. (4) In all 50 the shared arm changes the observation. (5) In all 50 it leaves the page on the evaluator's route. (6) In all 50 BrowserGym's own reference recovery leaves the original URL unrestored. (7) In all 50 the isolated arm preserves the baseline observation exactly. (8) In all 50 the two arms return equal grades. (9) At least 15 obtain a positive shared grade, which guards against a census made entirely of vacuous failures. No threshold was changed after outcomes were seen.

Result. All nine pass, with 17 positive shared grades against the threshold of 15. The recovery step restores its Python page references and never navigates, so the URL stays where the check put it. An independent re-derivation of the counts from the records agrees. Gates (4) and (5) follow from the selection of targets that must navigate, so the census's information is in (6), (7) and (8).

Limits of the test-page census. The pages are local read-only copies rather than the remote deployment, so the numbers concern the adapter's control flow, not live site behavior. Each template contributes one target rather than a complete task score, so this says nothing about how many of WebArena's published labels are wrong. Page isolation protects browser state and does not snapshot a shared server backend, so the disposable-page result covers this page-local channel only.

G.1 Live follow-up: two evaluators, one failed competence gate

A later pre-registered protocol (`webarena_live_prereg_20260925.md`) ran Llama-3.3-70B-Instruct (FP8, temperature 0) on all 66 shopping-admin tasks whose evaluator has a fixed URL target. Its two primary arms grade with the shipped validator or on a disposable page, with the same prompt, ten-step limit and site image. The first 15 task IDs, fixed by the launcher, also received one repeat per arm, on a different site instance from their primary episodes. Site databases were reset before each episode, and four independent site instances ran the final cohort; earlier reset failures are retained separately and excluded under a recorded amendment.

The evaluator in the first run. The site runtime installed the packaged `libwebarena` evaluator, which opens a disposable page, so in this run both arms were page-isolated by construction and the shipped arm tested the adapter as a PyPI user gets it. The episode logs exposed this: 75 of 617 observations under the shipped arm and 79 of 597 under the isolated arm were of the evaluator's target page, the agent's own visits in both. Amendment 4 then reran the four arms with the upstream behavior swapped in, the same file with its disposable-page block removed so that `goto` acts on the page the evaluator is handed, and nothing else changed.

Results with the packaged evaluator. All 162 declared records, 66 primary pairs and 15 four-arm comparisons, completed without episode-level infrastructure errors. Each primary arm succeeds on 5 of 66: four successes are shared, one occurs only under shipped grading and one only under isolated grading. Gate G2 (at least 50 complete primary pairs) passes, but G3 (at least 10 isolated successes) fails. These counts do not certify the absence of a grading effect.

On matched tasks the intervention difference is not separated from repeat variation (Table 15). A post-hoc diagnostic finds that 108 of 132 primary episodes end at the step budget, with 335 action errors over 1,214 actions, of which 57 are parser rejections of empty normalized actions. The model's full replies and per-action error messages were not retained, so the empty actions cannot be attributed to the model or to first-line normalization, and the other 278 errors stay unexplained. Constructed fenced replies expose a normalization defect, but a task-free eight-call probe of the model returns eight parseable replies. The diagnosis therefore stays mixed across the agent, the action in-

Table 15: Live WebArena follow-up, packaged-evaluator run. All comparisons below use the same 15 tasks; the main 66-task effect is not compared to a 15-task floor.

comparison	passes, left	passes, right	disagree
shipped versus isolated, first runs	2	1	1
shipped versus isolated, repeats	0	0	0
shipped first versus shipped repeat	2	0	2
isolated first versus isolated repeat	1	0	1

terface, observation and budget limits and site variability. We neither lower the gate nor count this run as evidence of an outcome effect.

Results with the upstream evaluator. The rerun produced 162 episodes. Three shipped episodes crashed when the evaluator's navigation of the agent's page timed out, and none on a disposable page, so 64 tasks have both primary arms. The page moves as the source predicts: 552 of 602 shipped-arm observations are the evaluator's target page (92%), against 93 of 594 on a disposable page (16%), and 130 of 140 against 62 of 150 on the repeats. The shipped arm passes 3 of the 64 tasks and the disposable page 5, losing 2 and gaining none, and both lost tasks pass in both arms of the packaged-evaluator run and in this run's isolated arm. The repeats disagree with their first runs on 0 of 14 shipped and 1 of 15 isolated tasks, and the isolated arms of the two runs, which grade the same way, agree on all 66. The agent answered on 2 shipped and 7 isolated tasks, and 59 and 54 episodes ended at the step budget. Gate G2 passes and G3 fails again, 5 isolated passes against 10 required. Amendment 4's first prediction, that most shipped observations are the target page, holds. Its second, that recorded scores fall under shipped grading, fits the 2 lost tasks but is not established by this sample. We report the disturbance as measured and the outcome effect as untested.

H THE SNAPSHOT REMEDY THE BENCHMARK SHIPS

AppWorld serves two endpoints that look like the repair this paper argues for: `/save_state` writes a checkpoint, and `/load_state` restores one after calling `AppWorld.close_all()` and replaying the preamble. Snapshot, grade, restore is exactly the shape of a non-interfering grading call.

Snapshots prevent the damage. A fourth arm of the HTTP experiment issues `/save_state`, then `/evaluate`, then `/load_state` after every action. It changed the official evaluation on none of the 54 families that completed every arm, against 7 for the arm that grades without restoring, so on this benchmark snapshot, grade, restore prevents every case we observed.

The world then cannot be closed. The same arm failed at `/close` on all 55 families it was run on, and the control and interleaved arms on none. The cause is not the grading call. `load_state` calls `close_all()`, which stops the world's time freezer, and `close()` later stops it again. `freeze_time.stop()` (`common/utils.py:4065-4067`) checks that its handle is not None but never clears it, so the second stop reaches a freezegun object that has already been torn down. The world's state is not at risk: AppWorld saves it at the end of every `execute` (the `_save_state` call in `environment.py`), so the failing stop follows the last save and the offline evaluation of that arm reads a complete world. Every sequence that raises contains `load_state`, which both stops and restarts the freezer, so the runs cannot separate a non-idempotent stop from an unpaired start; the reading is a source-level diagnosis, not a measured one. Table 16 isolates the sequence, one case per fresh process, two of them with no grading at all.

Table 16: Which sequences of shipped calls leave the world closable.

sequence	outcome
`execute` → `close`	closes
`execute` → `evaluate` → `close`	closes
`execute` → `save_state` → `load_state` → `close`	raises
`execute` → `save_state` → `evaluate` → `load_state` → `close`	raises

Any use of `load_state` leaves the world unclosable, and the evaluator is not involved. The repair is one line, clearing the handle after stopping so that the stop becomes idempotent, but it changes the benchmark rather than how a user calls it. Until it is made, the only non-interfering option we found through the unmodified public HTTP endpoints is to grade after `close()`, never during. A user who runs the grader in process can instead fork it (§5).

I Environments that can hold this hazard

The condition of §2 needs a grader that acts on live state and a later read of what it writes, either an action after a mid-episode call or another grading call after the end. Most environments have no such grader, so the denominator for any prevalence claim is much smaller than a list of benchmarks. We read the sources of seventeen environments from sixteen packages, the last two being τ-bench and its successor, locating environment classes structurally and classifying them by hand (Table 17). Three earlier attempts to classify by pattern matching each gave a wrong answer, since a regex cannot tell a task grader from an object-detection evaluator, a Settlers-of-Catan engine method or a static property of a game definition.

Table 17: Whether a package exposes a grader callable on live state, and what a source audit finds reachable from it. A *path* is a grader that acts on live state and a later read of what it writes. Verdicts cover this path only.

package	separate grader on live state	verdict
AppWorld	yes, public method and HTTP endpoint	path; interference measured
BrowserGym–WebArena	yes, live page passed to the grader; the upstream evaluator navigates it, the packaged `libwebarena` port opens a disposable page	upstream: path, page moved; packaged: no path
τ-bench	yes, reward method on the live environment	path; interference measured
ScienceWorld	yes, over py4j into a shared JVM	path; unresolved by a single-run comparison
ALFWorld (3D) (Shridhar et al., 2021)	yes, in the simulator variant	path; not tested, needs a GPU simulator
SWE-bench	no, tests the patch in a fresh container and grades the log	no path
τ^2-bench	no, scores the gold actions in a freshly built environment (`evaluator_env.py`, release 1.0.1)	no path
Inspect	scoring state kept, scoped per task	no path
eight environments, listed below	no, the reward is a return value of the transition function	no path
Overcooked-AI (Carroll et al., 2019)	no, scores recorded trajectories	no path

The eight environments are ALFWorld in its text version (Shridhar et al., 2021), TextWorld (Côté et al., 2018), MiniWoB (Liu et al., 2018), Crafter (Hafner, 2022), Minigrid (Chevalier-Boisvert et al., 2023), gym-sokoban (Schrader, 2018), TextArena (Guertler et al., 2025) and Gymnasium (Towers et al., 2024). SWE-bench has no path by construction: its harness applies the submitted patch in a freshly built container, runs the tests there and scores the printed log, and nothing later acts in that container. Inspect (UK AI Safety Institute, 2026) keeps process-global state around scoring, a dictionary of in-flight scoring passes, but keys it by task and cancels it at run boundaries. Inspect and AppWorld both hold process-global state near the grader, and only Inspect scoped it.

J An environment whose floor swallows the effect

ScienceWorld exposes a grader callable on live state, and its source draws the distinction our experiment needs: it marks `look()`, `inventory()` and `taskdescription()` as actions that consume zero time, and leaves `get_goal_progress()` outside that block. We replayed the benchmark's own gold action sequence for 30 tasks at 2 variations in four arms, each in its own process since the environment owns a JVM. The arms are a baseline, a repeat of the baseline, one

that calls the zero-time `look()` after every action, and one that calls `get_goal_progress()` after every action.

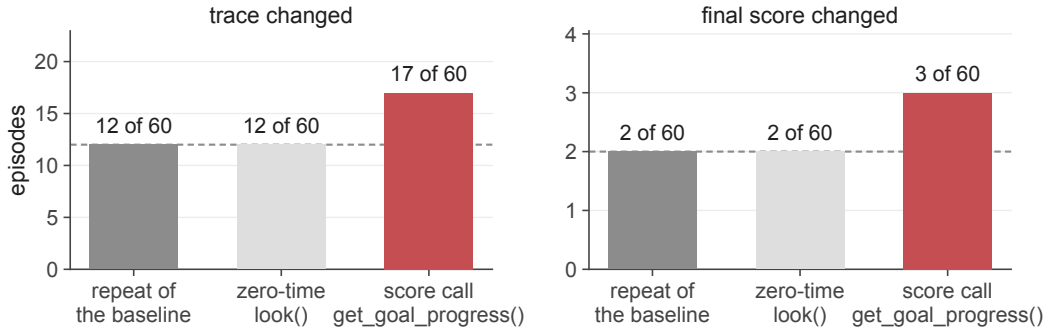

Figure 11: **ScienceWorld's repeat floor is close to the effect.** Episodes of 60 whose trace or final score differs from the baseline's. The dashed line is the repeat of the baseline, which differs from it in nothing.

The score call changed the trace in 17 of 60 episodes and the final score in 3 (Figure 11). The zero-time call changed the trace in 12, and so did the repeated baseline, which differs from the baseline in nothing. ScienceWorld does not replay deterministically across JVM runs, and against a floor of 12 a differential test on single runs cannot resolve an effect of this size.

We therefore report ScienceWorld as unresolved rather than as clean. A differential test on single runs reads evaluator interference only where the replay floor lies well below the effect. Repeated runs per arm could estimate a stochastic environment's floor, at a cost we did not pay here. AppWorld's floor, measured with the same kind of arm, is zero over 165 rows, which is what makes 18 of 165 readable at all.

K Proof that a forked grader reaches the parent's memory only through files

Theorem 1, restated. Let the grader run in a child created by `fork()`, returning its result over a pipe, with its address space discarded, and suppose every shared mapping the parent holds at the moment of the fork is backed by a named file. Then a store the grader makes to its own address space reaches the parent only through the pages of such a file, and the grader can influence the parent only through the file system, inherited descriptors, or state outside both address spaces.

Proof. After `fork`, POSIX specifies that the child is a separate process holding its own address space, initialized as a copy of the parent's. Implementations realise the copy lazily for private mappings: a page is shared read-only until one side writes to it, and that side then receives a private copy, so the child's store leaves the parent's view of the page unchanged. This holds for every private mapping, including one the child makes writable with `mprotect`, and for no shared one. A region mapped `MAP_SHARED` is neither copied at fork nor copied on write, so a store through one is visible to the parent at once. The child can also raise the protection of such a region after the fork whenever the object behind it allows, so protection at the fork bounds nothing, and the condition concerns what backs the region instead. A shared region backed by a named file is the page cache of that file, so a store through it lands in the file's pages. Within the file's length it changes the file. In the tail of the last page beyond the end of the file it changes nothing a read of the file returns, although every process that maps the page sees it, and the conclusion counts both as the file's pages. Anonymous shared memory, System V segments and `memfd` regions have no such file, and the condition excludes them. Under it, every mutation the grader makes to Python objects, module globals or class attributes is a store to a private mapping and is invisible to the parent. Mappings the child creates after the fork are private to it unless both processes map the same object, which is then a file or state outside both address spaces. What the parent can still observe are the objects the two processes hold in common once the address spaces are separate. These are the file system, including

the files it maps, the descriptors open at the moment of the fork, and state that was never in either address space. □

The condition as measured. The isolation run read `/proc/self/maps` in the parent before every fork. Across 5,375 forks no shared region was writable and at most 13 were read-only. A diagnostic replay of one census trajectory with the same parent finds 13 at every fork: SQLite's memory maps of the twelve base database files and glibc's character-conversion cache. In a forked child, `mprotect` makes each database map writable and fails with `EACCES` on the cache, which glibc opens read-only. Each database map covers its whole file, a whole number of 4,096-byte pages, so these maps hold no byte past the end of a file. The cache's map extends 1,670 bytes past its file, but the child cannot make it writable. A child could therefore store into a base database through its map only by changing the file, and the parent would see the store without a read. That is why the base-database directory is hashed around every call, and across the 5,375 calls it never changed. Asking only that no shared mapping be writable at the fork would not exclude such a store, and a content hash alone would not see a store past the end of a mapped file.

Two operating assumptions. As we run it, the construction also relies on the parent waiting for the child, so the two never act on a shared open file description at the same time. It also relies on the child leaving through `os._exit`, so no exit handler or buffered write inherited from the parent runs in it. Both are properties of the harness rather than of `fork`, and a pipelined or asynchronous forked grader would have to establish them again. A parent with other threads can leave the child holding a lock that no thread will release, which can hang a forked call but cannot carry a store into the parent. The descriptor channel is not hypothetical here: AppWorld caches SQLite connections opened with a memory-map pragma, and the child inherits their descriptors. Two practical limits follow. The cost of `fork` grows with the parent's resident memory, since page tables are copied, and a process that has initialized CUDA cannot fork safely, so a pipeline that hosts the model inside the environment process cannot use this construction as it stands.

Scope of the three channels. Read narrowly they are not exhaustive, so we read them broadly and say how. The file-system channel is not only writes to file contents: creating, removing, renaming or truncating a path, and changing permissions or timestamps, all reach the parent. The audit hook sees all of these, and the directory hash sees those that change content; a change to metadata alone is outside what the hash claims. The descriptor channel is not only a shared file offset. Locks held on an open file description, which is what `flock` and OFD locks are, are shared with the child, so a child that releases one releases it for the parent; classic POSIX record locks are held per process and are not inherited. The third channel is a residual rather than an enumerated item. It covers sockets, subprocesses, System V and POSIX message queues, signals, the child's exit status and the `SIGCHLD` its exit delivers, and kernel interfaces that write another process's memory directly, such as `ptrace`, `process_vm_writev` and `/proc/<pid>/mem`, which a grader would have to invoke deliberately. We measure the file-system channel and the mapping condition but not the descriptor channel, which §5 leaves open.

L AN IN-MEMORY FINGERPRINT THAT DID NOT CERTIFY NON-INTERFERENCE

Before settling on the fork guarantee, we built the fingerprint that §5 mentions, which would have made that guarantee unnecessary. It hashes every module-level and class-level mutable binding in the environment package, 1,576 to 1,577 roots in a live AppWorld process, before and after each grading call. If the fingerprint never moved, determinism would imply, by a one-line induction over steps, that the recorded outcome did not move either.

On two smoke-test tasks, neither of which shows interference, the fingerprint changed at all 14 and all 21 of their grading calls. SQLAlchemy rebuilds column descriptors and class managers when the evaluator constructs its own model collection, and a canonicalization conservative enough to catch a rebinding must count a rebuilt object as a change. The largest single fingerprint also hit its depth limit 57,563 times, so the fingerprint was at once too sensitive to be specific and too shallow to be complete.

We abandoned it, since a check that fires on every call supports a theorem whose hypothesis never holds. Its failure is why we guarantee how the grader runs (§5) rather than certify what it touched. A finer canonicalization might do better, and we did not build one.

M PRE-REGISTERED PROTOCOLS, THEIR OUTCOMES AND WITHDRAWN CLAIMS

This appendix first lists every gate and prediction we declared before its outcome was read (Table 18), then the claims we withdrew. Six gates failed, and so did two predictions and part of a third, the count §7 gives.

Table 18: Every gate and prediction declared before its outcome was read.

protocol	declared before any outcome	outcome	reported in
τ-bench replay census	gates (1), (2), (4): repeats agree, forked arms return the ungraded reward, paired arms grade at the same steps	pass	App. C.2
	gate (3): at least 1,000 eligible trajectories	**failed**: 153, so the census is exploratory	App. C.2
τ-bench wrapper, end to end	every excess pass falls on a task without an output check	**prediction failed**: 100 of 105	§3.1, App. C.1
AppWorld published runs	G1: at least 80% of a run's tasks eligible	**failed** for 3 of 4 runs	§3.2, App. E
	G2, G3: the repeated ungraded arm and every forked arm change no eligible evaluation	pass	App. E
	in-process grading destroys successes and manufactures none; the script rewrites downward; forked arms change none	held where the records can check it	App. E
WebArena test-page census	nine gates on 50 templates	all pass	App. G
WebArena live, packaged evaluator	G2: at least 50 complete primary pairs; G3: at least 10 isolated successes	G2 pass; **G3 failed**: 5	§3.3, App. G.1
WebArena live, upstream evaluator (Amendment 4)	the same G2 and G3	G2 pass; **G3 failed**: 5	§3.3, App. G.1
	most shipped observations are the target page; scores fall under shipped grading	first held (92%); second untested	App. G.1
live ACE, second model	flagged families move; cleared ones do not beyond the floor; no success is manufactured	first and third held; **second failed**: 2 of 51	§4.1, App. B
AppWorld family predictor	primary gate	**failed**: infeasible as drafted	App. M
fork overhead	a forked call within 1.5× the in-process median	**failed**: 262 against 113 ms	§5
live attribution, two cleared live families	the same-actions family is the tracker channel; the other changes on each arm's fixed sequence	**first held; second failed**: the change runs through the agent's next observation	§4.1, App. B
held-out restoration check	in-process grading changes only the 5 known events; `home_path` repairs the 3 with unchanged records, the other six the other 2, all seven all 5; no other row changes	all five held	§4.3, Table 19

A pre-registered test that returned infeasible. We pre-registered a protocol predicting which families would be affected, with its population, gates and minimum event count fixed in advance, and ran it over 139 family representatives from `test_challenge`, a split no earlier experiment

had touched. The primary gate did not pass, for a reason of ours rather than a property of the prediction. One feasibility clause required every forked arm to reproduce its ungraded arm, which eligibility also requires, so the gate could not pass whenever any task was ineligible. The five rows that broke it had their arms assigned to different tasks by the shard collision of Appendix N. We report the verdict as infeasible and do not restate the gate after the fact. The registered claim, zero false negatives with at least three interference events, would have failed on its own terms as well: of the run's 5 events on 139 rows, the predictor, which named 7 rows, caught 4 and missed 1, with 3 false alarms. The run's secondary observations are exploratory, and the strongest is that the unchanged-records signature recurred on three tasks in that population.

A held-out restoration check. To ask whether the two channels of §4.3 account for events elsewhere, we fixed a prediction in advance (`HELDOUT_RESTORATION_PREREG_20260926.md`, SHA-256 `7524f722...`) and then ran the restorations on the 130 eligible rows of that held-out run. The worker is the one of §4.3, at that run's cadence of one call before the first action and one after every action. The run's five events were known beforehand, and no restoration had run on these rows. In-process grading again changes exactly those five (Table 19). Restoring `home_path` repairs exactly the three whose saved records are byte-identical, one of them the event the registered predictor missed, and restoring the other six repairs exactly the other two. Restoring all seven repairs all five, and no restoration changes any of the other 125 rows or manufactures a success. All five predictions held. Because the events were known, this is evidence that the two channels transfer, not a test of the predictor, which is unchanged.

Table 19: The restorations of Table 8 on the 130 eligible held-out rows, run after a prediction fixed in advance. Events are grouped by the records signature recorded before any restoration ran, and repaired or unchanged rows are judged as in Table 8.

	damaged rows repaired		other rows	successes
restored after every call	records unchanged	records changed	changed	destroyed
nothing (in-process grading)	0 of 3	0 of 2	0 of 125	4
`home_path`	3 of 3	0 of 2	0 of 125	2
the other six	0 of 3	2 of 2	0 of 125	2
all seven	3 of 3	2 of 2	0 of 125	0

Claims withdrawn during this work. Table 20 gives the four claims withdrawn during the work and the seven withdrawn after internal review, each as first made, with why it was withdrawn and what the paper says now.

N COLLISIONS IN OUR OWN RUNS

Every census number in the body comes from the defect-free re-census, the last column of Table 9. This appendix records two collisions in our own experiment directories that shaped the earlier columns. Neither changed a reported number, and repairing the first is why the design of §2 carries a fourth arm.

The census worker's original read of the counter file materialized entries in a `defaultdict`, and those entries were then persisted, so the graded arms wrote a file the ungraded arm never could. We re-ran all arms with a read that cannot materialize. That re-run was sharded, and each shard numbered its own subset of tasks from one while writing its scratch specification into a shared directory under a name built from that index. Six shards per trajectory set ran at once and overwrote one another's specifications, so a worker could replay a task other than the one the manager was assembling a row for. The `finally` clause that removed each specification after use also deleted other shards' files in flight, which crashed six workers.

Thirteen of 159 completed rows compared arms belonging to different tasks. The experiment directories are keyed by task and were not cross-contaminated, and every arm records the task it actually ran, so a row is sound exactly when all its arms agree on task identity. On the 146 sound rows, the eligible set, the event count, the direction of every changed success, the count of grade changes with identical records and every detector cell equal their values with the contaminated rows included, because all thirteen were already ineligible. The defect did produce one alarming artifact: on the

Table 20: Claims withdrawn during this work, and what replaced them.

as first made	why it was withdrawn	what the paper says now	where
during the work			
the binding predicate is sound	it missed an event in the intermediate re-run (Table 9)	the least lossy of four detectors, and nothing more	Fig. 7
AppWorld replay is not bit-reproducible	inferred from the contaminated rows of the first collision	two replays agree on all 165 rows	App. N
the snapshot endpoints are a working remedy	they prevent every case we tested and then leave the world unclosable	through the public endpoints, grade after `close()`	App. H
the unchanged-records signature grew from one task to two inside the census	seen only in the re-run that carried the shard collision	one task, three rows, in the defect-free census (Table 9)	App. N
after internal review			
a single grading call is safe	the placement sweep finds no change after the last action but changes at earlier placements in this sample	safe when nothing that reads the written state remains	§4.2
the counter key is the mechanism	it accounts for one affected family in seven, and restoring it alone repairs 3 rows of 18	two channels; the tracker and its handles carry the other 15	§4.3
the error cannot run the other way	both channels lose writes, and a lost write can satisfy an assertion that a change be absent	the one-sidedness is observed rather than guaranteed	§4.3
the mapping probe as first drafted	it failed its own first control, because Python's `mmap` defaults to `MAP_SHARED`	fixed before any reported run	App. B
the audit hook records every write-mode `open`	it tested only the builtin's mode string and missed every `os.open`	hook corrected and the isolation run repeated	§5
a 4.4% bound on manufacturing a success	computed over 67 rows, of which only 7 carried an interference event, and no sampling model would justify a population bound	exact counts, no population bound	App. A
an outcome-level Fisher test ($p = 8.7 \times 10^{-5}$) shows the direction	a changed evaluation on a row that passed every test is a failure by definition, so half of its table was fixed in advance	the direction rests on assertion counts	§4.3

contaminated rows the forked control appeared to disagree with the ungraded arm thirteen times, which we briefly read as evidence that replay is not reproducible. On sound rows that disagreement is zero.

The replacement gives each shard its own specification directory and refuses to emit a row whose arms disagree on task identity, so a future collision fails loudly instead of producing a plausible cross-task comparison. It also adds the repeated ungraded arm: on its 165 rows, two independent replays of the same trajectory produce the same official evaluation every time, and so does the forked arm.

A second, smaller collision. The corrected isolation run of §5 repeated a smaller version of the mistake. Its names carried no run token, so the first task of every shard reused a directory the original run had written, and a side run with a different shard count wrote into one of those directories at the same time. That cell is the only one whose outcome moved in the first pass, and the side run's own cell there is the only one whose hash caught files other than the grader's report. Both anomalies came from two processes sharing a directory, not from the fork: the cell did not move in three reruns under fresh names. We reran under fresh names all 123 cells whose names could collide, and we report those reruns; none moved.

The same collision in the held-out run. The held-out predictor run of Appendix M used the sharded worker of the first collision. Five of its 139 rows compared arms of different tasks, and two

more crashed on the specification file the shards shared. All seven are ineligible, so none enters an interference count, and each of its 130 eligible rows compared arms of one task. The five are the rows that broke that run's feasibility clause.

O Related work

Graders that are wrong, and graders that change what they grade. Scoring by running code (Trivedi et al., 2024a; Zhou et al., 2024; Jimenez et al., 2024; Wang et al., 2022) makes the grader a program with side effects. Audits have found contamination, metric-dependent gains, construction defects and wrong verdicts (Oren et al., 2024; Yue et al., 2025; Zhu et al., 2025; Cemri et al., 2025; Dong et al., 2026). These ask whether one verdict is right; we ask whether a correct verdict changes the next one. BenchJack treats a grader that shares state with the agent as a way for the agent to cheat (Wang et al., 2026), the reverse of our direction. Evaluation-aware models (Needham et al., 2025) show a behavioral observer effect, not a mechanical one.

Existing practice that preserves state. Tree search scores partial trajectories in the live environment (Koh et al., 2024), and agentic RL frameworks take per-turn rewards from the transition function or a terminal grade (Wang et al., 2025; Feng et al., 2025; Xi et al., 2025). Several systems design around the hazard. VAGEN restricts its verifier to read-only actions (Cui et al., 2026), ProRe replays actions on its own emulator (Dai et al., 2026), ToolSandbox scores per-turn database snapshots (Lu et al., 2025), and the Interactive Reward Agent notes that its tools can change what they inspect (Shi et al., 2026a;b). OdysSim's τ-bench runtime scores a copy, so repeated reward requests cannot turn a failure into a success (Zhou et al., 2026). We state the condition these designs protect, show that shipped code breaks it and separate the mechanisms behind it.

Connections to earlier notions. Non-interference comes from security (Goguen & Meseguer, 1982). Self-composition checks it by running a program twice (Barthe et al., 2004), as our design does, and secure multi-execution enforces it with a copy per level (Devriese & Piessens, 2010), which our forked grader resembles. In testing, pollution between tests was assumed away until measured (Zhang et al., 2014), and rerunning separates order-dependent tests from flaky ones (Lam et al., 2019), as our repeats do. One run can also change the next run's starting state (Casheekar & Tangirala, 2026), which our cross-boundary test on AppWorld does not find. The probe effect (Gait, 1986; Mytkowicz et al., 2009) and reward tampering (Everitt et al., 2021) are the closest known failure modes.