

Zero-Call Verification: End-State Agreement Outperforms Claude Opus 5.5 Judges on τ^2 -bench and Compiled Checks Outperform an ACE-Style Verification Playbook on AppWorld

Lindell S. Cumes

QWNTL Labs

Project page: <https://qwntl.com/paper-verification-capital>

Abstract

Agent harnesses check a tool agent’s work by asking a model (an LLM judge, the agentic verifier VeriHarness, a critic) or by running tests a model wrote once per task (HarnessPro). Claude Opus 5.5, checking its own work, accepts 65 of 66 failed attempts on the AppWorld tasks it fails in every repeat. We introduce *zero-call verification*, two done-checks that make no model call per checked attempt: end-state agreement flags an attempt whose final environment state is not the most common among other attempts at the task, and compiled checks are executable specifications written per task template from past pass/fail outcomes under a negative-selection gate. On 600 Claude Opus 5.5 attempts at held-out τ^2 -bench tasks, end-state agreement catches 48.5% of failures at 1.3% false fire with no labels, against 30.9% for a Claude Opus 5.5 judge shown the same attempts (preregistered gain of 16.5 points in Youden’s J [+5.4, +32.1]). On 31,434 AppWorld attempts, 449 of them from smoke-test and no-op runs, compiled checks for 48 templates catch 90.1% of never-seen failure states and fire on 0.026% of correct attempts, where an ACE-style verification playbook catches 55.5% at matched false fire on twenty times the checks’ tokens. Checks compiled only from other models’ failures verify Claude Opus 5.5. Frozen checks depreciate, losing coverage through forgone revisions even when the library holds every template’s check. Checks and verdicts are released.

1 Introduction

An agent that finishes a task leaves an end state: the airline database after a τ^2 -bench customer-service chat, or the artists an AppWorld Spotify account follows. Whether that end state is correct decides whether the attempt counts. The decision belongs to the *harness*, the code that supplies a model’s context and tools and decides when its work is done.

Benchmarks decide with a grader that knows the goal state. A customer-service agent outside τ^2 -bench has none, and an agent that works unattended must decide at every finished task.

The current answer is to ask a model. LLM-as-a-judge grades an output [41] and Agent-as-a-Judge a trajectory [42]. VeriHarness has the generating model verify a pool of its own rollouts against workspace evidence, and raises Claude Opus 4.8’s mean score on five workspace benchmarks by 6.4 points over a single rollout [36].

Rethinking Verification in Agent Harnesses finds that an independent critic adds 5.9 to 9.5 points on Terminal-Bench at 1.13 to 1.25 times the baseline cost, and supports independent review as a practical axis of test-time scaling [39]. HarnessPro, which finds LLM judges “overly optimistic” on execution trajectories, scores candidates with tests that a model writes once per task and keeps fixed for the whole run [27].

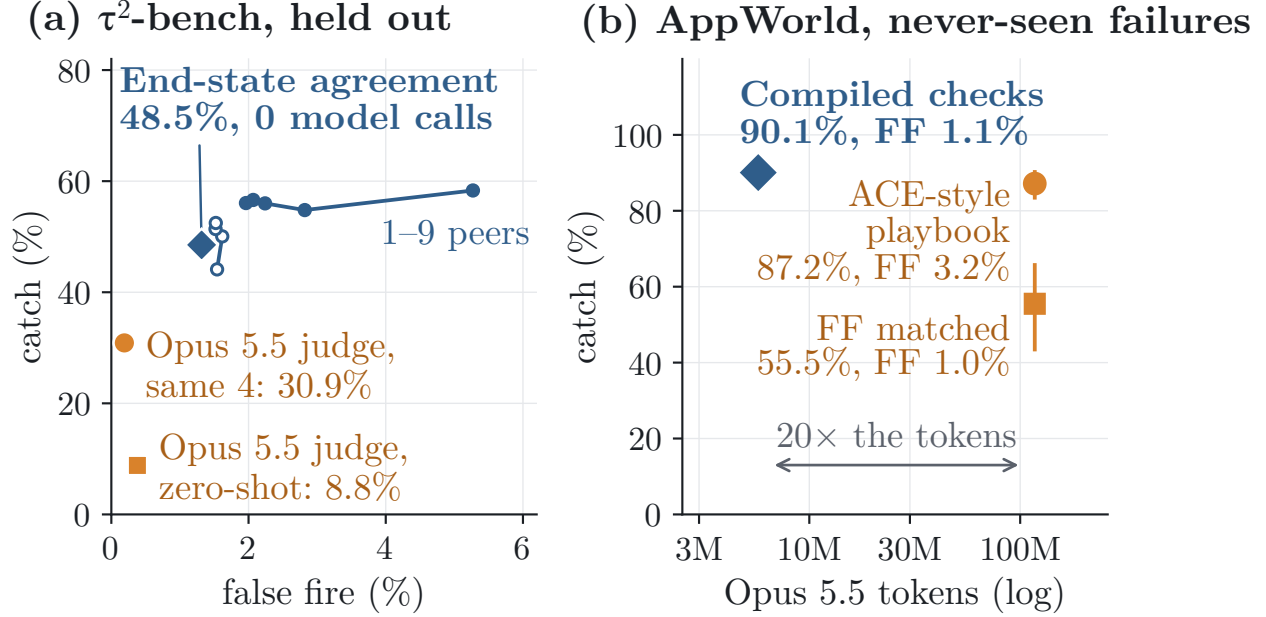


Figure 1: **Zero-call verifiers catch more failures than Claude Opus 5.5 judges and an ACE-style verification playbook.** (a) Held-out τ^2 -bench: 600 Claude Opus 5.5 attempts at 60 tasks, 68 of them failures; catch against false fire, both per attempt. End-state agreement traces its budget curve over $k = 1, \dots, 9$ peer attempts (mean of 50 seeded panels; odd and even k joined separately); the diamond is the preregistered panel of four. Each judge makes one Claude Opus 5.5 call per attempt; the peer judge sees the same four attempts as agreement. (b) AppWorld: catch of 1,036 never-seen failure states in windows 5–38 against Claude Opus 5.5 tokens spent over the stream; false fire (FF) is per never-seen correct state. Compiled checks run without a model call; the ACE-style verification playbook judges each attempt and reflects and curates after each window. Bars are 95% template-cluster bootstrap intervals.

Judges and critics pay a model call at every check, and they carry the blind spot of the model they call. On the 22 AppWorld test-challenge tasks that Claude Opus 5.5 fails in all three repeats, Claude Opus 5.5 checking its own work accepts 65 of the 66 failed attempts. A fresh Claude Opus 5.5 judge shown only the clean state diff accepts 64 (Appendix A.2).

Asking again does not help. For a Claude Haiku 4.5 verifier on 39 failed AppWorld trajectories that claimed completion, false acceptance is locked to the item, with an intraclass correlation of 0.63. Three samples cut it only from 0.64 to 0.51, where independent checks would reach 0.26; the fitted law puts 100 re-checks at the value of about three independent checks (Appendix A.2).

A judge must perform three inferences: reconstruct what the task required, read what the attempt changed, and decide whether the changes meet the requirement. A failure that omits a required change leaves nothing in the change list to read. A failure that takes a plausible default where the task required another value reads as compliant.

Zero-call verification replaces each inference with an equality. End-state agreement compares an attempt’s final environment state with the final states of k other attempts at the same task, and flags it when its state is not the most common one. It needs no label and no model call, and it reuses attempts the harness already records: retries, parallel workers and other models. It works because correct attempts at a task coincide while failures scatter.

Compiled checks compare the final state with a requirement that a deterministic Python function re-derives from the instruction and the start state. Claude Opus 5.5 writes one check per task template from past pass/fail outcomes. A negative-selection gate admits it only if it stays silent on the template’s past successes, and the check then runs on every later attempt without a model call (Figure 1).

On 600 Claude Opus 5.5 attempts at 60 τ^2 -bench tasks that no part of either verifier’s development used, end-state agreement with four other attempts catches 48.5% of failures at 1.3% false fire per correct attempt. Claude Opus 5.5 judging zero-shot catches 8.8%. Shown the same four attempts, the judge catches 30.9%, and agreement exceeds this

information-matched judge by a preregistered $\Delta J = +16.5$ points [$+5.4, +32.1$] of Youden’s J , catch minus false fire. Both saw the same evidence; agreement compared it.

On 31,434 graded AppWorld attempts replayed in order, a library that holds one compiled check for each of 48 templates catches 90.1% of never-seen failure states at 1.1% false fire per never-seen correct state, or 0.026% per correct attempt. An ACE-style verification playbook fed the same outcomes, with Claude Opus 5.5 as its judge, reflector and curator [38], catches 55.5% at matched false fire and spends twenty times the tokens.

A compiled check cannot see which model produced a state. In a run whose writer also saw grader reports, checks written only from Claude Haiku 4.5, Qwen 27B and GPT-6 Astra failures catch 30 of 32 never-seen Claude Opus 5.5 and Claude Opus 4.8 failure states, and fire on none of 7,185 correct attempts by those two models.

A check’s value also depends on its revisions. A copy of the library frozen after 200 failures catches a preregistered 24.0 points less than the library that keeps learning. Copies frozen after the library holds a check for every template still lose 12.0 and 10.6 points. Frozen checks depreciate.

This paper introduces zero-call verification and places each result against a named alternative. End-state agreement is the first done-check shown, in a preregistered information-matched comparison, to catch more of a frontier agent’s failures than the same frontier model judging them.

VeriHarness argues that voting, like LLM judging, relies “on what the model already believes rather than on evidence the environment could provide” [36]. Voting over end states is voting over environment evidence, and it exceeds the information-matched judge’s J at every panel size from one peer to nine.

Compiled checks improve on an ACE-style verification playbook by 34.6 points of catch at matched false fire, on a twentieth of the tokens. HarnessPro keeps its generated tests fixed [27]; this paper gives the first measurement of what a frozen verifier loses as an ordered record moves on. VeriHarness compares frozen and evolved skill libraries only on a fixed held-out split [36].

2 Problem Statement

A task instance is $T = (d, s_0, V)$: an instruction d , a start state s_0 of the environment’s database, and an official verifier V . An agent $A = \langle M, H \rangle$ pairs a model M with a harness H . An attempt a by A at T leaves an end state $s(a)$ and a submitted answer $\hat{x}(a)$, and the official verifier returns $y(a) = V(a) \in \{0, 1\}$, with $y = 1$ for correct. Attempts at the same task recur: retries, parallel workers, other models and other harnesses.

A *done-check* returns a flag $f(a) \in \{0, 1\}$ from what the harness holds, and must not read $y(a)$ or V . Catch is $\Pr[f = 1 \mid y = 0]$ and false fire is $\Pr[f = 1 \mid y = 1]$. Youden’s index $J = \text{catch} - \text{false fire}$ [34] is reported as a fraction, and differences ΔJ in points ($\times 100$); a ΔJ is not a difference in catch.

The budget is counted in Claude Opus 5.5 calls and tokens and has three parts: a one-time construction cost, a cost per checked attempt, and the number of attempts that reuse the construction. A judge pays one call per checked attempt and constructs nothing.

Zero-call verification is a done-check whose cost per checked attempt is zero model calls:

$$\max_f J(f) \quad \text{subject to} \quad c_{\text{check}}(f) = 0, \quad (1)$$

where c_{check} counts model calls made while checking one attempt. Writing a check is construction cost; Section 6.2 reports it against the rivals’ totals.

Information-flow contract. The attempts are a recorded stream, fixed before any verifier runs; no verifier changes what an agent does. Table 1 lists what each verifier reads, which labels it uses and when its verdict is fixed. Judges and the check writer are Claude Opus 5.5 with adaptive thinking at high effort, frozen throughout.

What persists across tasks differs: agreement keeps recorded end states, the check library keeps one admitted check per template, and a judge keeps nothing. Each verdict on an attempt is recorded before that attempt’s outcome is read.

Table 1: **Information-flow contract.** What each verifier reads when it checks attempt a , the labels it uses, its Claude Opus 5.5 calls per checked attempt, and what persists across tasks. “Past” means windows that closed before the window holding a .

Verifier	Reads when checking a	Labels	Calls per check	Persists across tasks
End-state agreement	fingerprints of $s(a)$ and of k other attempts at the task	none	0	recorded end states
Compiled checks	instruction, supervisor, start and end databases, submitted answer	past pass/fail outcomes (writing and gating only)	0	one admitted check per template
Zero-shot judge (τ^2 -bench)	policy, customer messages, database changes, relevant start records	none	1	nothing
Peer judge (τ^2 -bench)	the same, plus the database changes of the $k = 4$ agreement peers	none	1	nothing
Precedent judge (AppWorld)	rendered view of the attempt’s state changes, plus labeled earlier attempts of its template	past outcomes	1	labeled record
ACE-style verification playbook (AppWorld)	rendered view of the attempt’s state changes, plus the current playbook	past outcomes (reflector, curator)	1, plus reflect and curate per window	playbook text

3 Zero-Call Verification

3.1 The verification record

Attempts reach the harness in windows $W_1, W_2, \dots$. End states are fingerprints: on τ^2 -bench, $s(a)$ is a hash of the domain database after a strict replay of the attempt in a fresh environment; on AppWorld, it is a byte fingerprint of the attempt’s state changes. A *never-seen* state is a byte-distinct end state at its first appearance in the stream.

Definition 1 (Verification record). After window t , a harness holds $R_t = (E_t, Y_t, C_t)$: the end states E_t of the attempts in $W_1, \dots, W_t$, indexed by task; their official outcomes Y_t ; and the admitted checks C_t , at most one per template. A verifier built from R_t checks each attempt in W_{t+1} before any outcome in W_{t+1} is read.

End-state agreement draws on E_t alone and reads no label. A compiled check library turns E_t and Y_t into C_t with at most one model call per template per window, then checks with none. A precedent judge reads E_t and Y_t through one model call per attempt.

Novelty sets the denominator on AppWorld: 67% of failures repeat an earlier end state exactly, and 27,133 correct attempts reach only 843 distinct correct states. False fire per correct attempt and false fire per never-seen correct state therefore differ by more than an order of magnitude. Both are reported, each over the windows of the catch it accompanies.

3.2 End-state agreement

For an attempt a , let $P(a)$ hold k other attempts at the same task, and let $n_P(x) = |\{b \in P(a) : s(b) = x\}|$. Agreement flags a exactly when

$$f_{\text{agree}}(a) = \mathbb{K} \left[n_P(s(a)) < \max_x n_P(x) \right]. \quad (2)$$

The attempt is excluded from its own panel, an attempt tied for the most common state passes, and an empty panel passes (Algorithm 1).

Proposition 1 (What agreement sees). *Let a task have a single correct end state s^* , as each τ^2 -bench task does, and let $P(a)$ be non-empty. (i) A failure whose end state no peer reached is flagged. (ii) A failure whose end state is the most common state of its panel passes. (iii) A correct attempt is flagged only if some wrong end state occurs in $P(a)$ more often than s^* . (iv) A failure that leaves the end state s^* receives the verdict a correct attempt with the same panel receives; no end-state check separates the two.*

Each case follows from Equation (2): in (i), $n_P(s(a)) = 0$ while the maximum is at least 1; in (iii), $s(a) = s^*$, and only a state $x \neq s^*$ can exceed $n_P(s^*)$. Agreement therefore requires one condition: correct attempts coincide while failures scatter. A task’s failures may outnumber its successes, provided no single wrong state outnumbers the correct one in the panel. Case (ii) is a repeated mistake, and Section 5.4 measures how panels drawn from another model reach it.

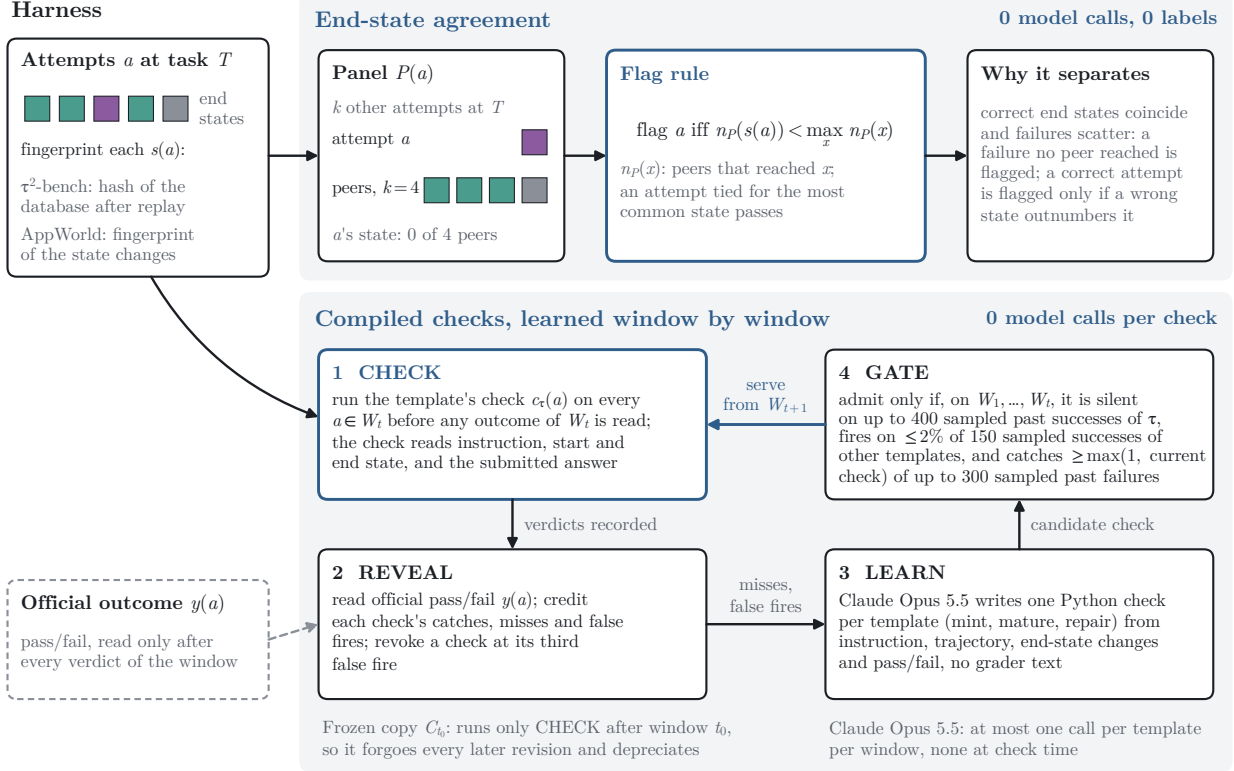


Figure 2: **Zero-call verification inside an agent harness.** Top: end-state agreement fingerprints each attempt’s final database and flags it unless its fingerprint is tied for the most common among k peers at the task. Bottom: the prequential check library runs four steps per window. CHECK records verdicts before any outcome is read; REVEAL credits catches, misses and false fires; LEARN has Claude Opus 5.5 write or revise one Python check per template; GATE admits a candidate only if it stays silent on the past successes it samples from its template. A frozen copy runs CHECK alone. Neither verifier calls a model when it checks an attempt.

3.3 Compiled checks

A check is a deterministic Python function of the task instruction, the supervisor’s identity, the start and end databases, and the submitted answer. It returns whether it applies, a flag, and a sentence or two of evidence with the values it compared. It reads neither the agent’s trajectory nor the actor’s identity, and it never sees the answer key (Appendix A.3).

The writer’s prompt instructs Claude Opus 5.5 to parse each instance’s parameters from the instruction, re-derive the correct outcome from the start state, and test the end state against it. A check is therefore an executable specification of one template, and it must hold for every instance of that template.

Definition 2 (Accrual and the gate). Window $t + 1$ adds its end states to E and its outcomes to Y . For each template τ with a failure in W_{t+1} that its check c_τ did not flag, the writer proposes a candidate c' . The gate admits c' into C_{t+1} only if, on attempts from windows $\leq t + 1$, it fires on none of up to 400 sampled successes of τ , on at most 2% of up to 150 sampled successes of other templates, and flags at least $\max(1, \text{flags of } c_\tau)$ of up to 300 sampled past failures of τ . A check is revoked at its third false fire after admission.

The gate is negative selection [17]. Weak-to-Strong Verifier Hardening admits a verifier repair by the same rule: among its gates, every known legitimate solution of the task, reference solutions included, must still pass on replay [35]. Here the rule admits model-written checks, and the record’s past successes play the legitimate solutions.

Proposition 2 (Revisions keep recorded coverage). *On the gate’s samples, an admitted revision of c_τ flags at least as many past failures of τ as c_τ did, and fires on none of the sampled past successes of τ .*

Algorithm 1: **End-state agreement for one attempt.** Zero model calls, zero labels.

Inputs: attempt a at task T ; recorded attempts $\mathcal{A}_T$ at T ; panel size k ; seed r .

- 1: $P \leftarrow$ a seeded sample of $\min(k, |\mathcal{A}_T \setminus \{a\}|)$ attempts from $\mathcal{A}_T \setminus \{a\}$
 - 2: **if** $P = \emptyset$ **then return** pass
 - 3: $n(x) \leftarrow |\{b \in P : s(b) = x\}|$ for each fingerprint x in P
 - 4: **return** flag **if** $n(s(a)) < \max_x n(x)$ **else** pass
-

Algorithm 2: **Prequential compiled-check library for one arm.**

Inputs: windows $W_1, \dots, W_{38}$; check writer M (Claude Opus 5.5); gate G (Definition 2).

State: library C with at most one admitted check c_τ per template τ ; $C \leftarrow \emptyset$.

- 1: **for** $t = 1, \dots, 38$ **do**
- 2: **CHECK:** for every $a \in W_t$, record $f(a) = c_{\tau(a)}(a)$ (pass if no check), before any outcome of W_t is read.
- 3: **REVEAL** $y(a)$ for $a \in W_t$; update each check’s catches, misses and false fires; revoke a check on its third false fire after admission.
- 4: **LEARN:** for each template with an uncaught failure in W_t , M writes a new check (*mint*) or rewrites c_τ from the uncaught failures (*maturation*); a false fire triggers a *repair*, and a candidate rejected by G is retried with the gate’s feedback.
- 5: **GATE:** admit a candidate only if G passes on items of $W_1, \dots, W_t$; admitted checks serve from W_{t+1} .

A frozen copy performs only **CHECK** after window t_0 . The anticipation arm also mints from up to two correct attempts of a template before its first failure.

The proposition is the gate condition read as a property of the library: on the gate’s samples, no admitted revision catches fewer of the template’s recorded failures than the check it replaces, and none fires on a recorded success.

Proposition 3 (Actor invariance). *Two attempts at the same instance with identical end states and identical submitted answers receive identical verdicts from every check.*

The proposition follows from the check’s signature, which excludes the trajectory and the actor’s identity. Transfer across actors therefore depends only on where a new actor’s failure states fall relative to the requirement a check re-derives.

Four operations change a library. A *mint* writes a check for a template that has none, from one missed failure. A *maturation* rewrites a template’s check from the failures it missed in the window. A *repair* rewrites a check that fired on a correct attempt, and a *retry* resubmits a rejected candidate with the gate’s feedback. One Claude Opus 5.5 call per template per window bounds the cost, and a check that fails to run counts as no flag.

3.4 Value and depreciation

Definition 3 (Value and depreciation). The value of R_t on a later window u is $\kappa_u(R_t)$, the catch of its verifier on the never-seen failure states of W_u , reported with its false fire on the never-seen correct states of W_u . A copy frozen at t_0 applies C_{t_0} to each window $u > t_0$. Its depreciation is

$$D_{t_0} = \kappa(R_{u-1}) - \kappa(R_{t_0}), \quad (3)$$

the difference in catch between the learning library and the frozen copy, pooled over the never-seen failure states of all windows $u > t_0$.

A frozen copy forgoes two kinds of accrual: checks for templates that first fail after the freeze, and revisions of the checks it holds. Once every template holds a check, the first kind is empty, and depreciation consists of forgone revision alone. Section 6.3 uses this split to locate the mechanism.

4 Evaluation Protocol

Records. Three recorded streams are used, each graded by its benchmark’s official verifier. The held-out τ^2 -bench split [2] holds 60 retail and airline tasks that no verification study in this paper had used; each task has one correct end

Table 2: **Evaluation protocol.** Every model role is Claude Opus 5.5 (claude-opus-5-5) with adaptive thinking at high effort and one sample; no verifier sees the official outcome of the attempt it checks. Matched axis: the same attempts per comparison (600 held-out τ^2 -bench attempts; 1,036 never-seen AppWorld failure states and 627 never-seen correct states in windows 5–38).

Arm	Model role, max tokens	Evidence and operating point	Calls
End-state agreement	none	$k = 4$ other Claude Opus 5.5 attempts; flag rule Equation (2); 50 seeded panels	0
Claude Opus 5.5 zero-shot judge	judge, 16,000	binary verdict	1 per attempt
Claude Opus 5.5 peer judge	judge, 16,000	binary verdict; peers’ changes capped at 4,000 characters each	1 per attempt
Compiled checks (pass/fail run)	writer, 32,000	binary outcomes, no grader text; gate Definition 2	261 writer calls over the stream; 0 per check
Precedent judge	judge, 16,000	$p_{\text{success}} \in [0, 100]$, flag at $p \leq \tau$	1 per attempt
ACE-style verification playbook	judge 16,000; reflector and curator 32,000	p_{success} , flag at $p \leq \tau$; Playbook, Reflector, Curator [38]	1 per attempt, plus 914 reflect and 914 curate

state. Its evaluation set holds up to 10 Claude Opus 5.5 attempts per task: 600 attempts, 68 failures and 532 correct, from 58 runs of 13 harness configurations (Appendix A.1).

The τ^2 -bench training record holds 1,133 attempts at 104 tasks by six actor models, 618 of them failures. The AppWorld record [31] holds 31,434 graded train and dev attempts on 48 templates with three instances each, 3,607 of them failures, forming 1,186 failure states; 449 of the attempts come from smoke-test and no-op runs.

Actors. On AppWorld the actors are Claude Haiku 4.5, Claude Opus 5.5, Claude Opus 4.8, GPT-6 Astra and two open-weight 27B models, Qwen3.5-27B and an INT4 Qwen3.8-27B, pooled below as Qwen 27B. Most failures are Claude Haiku 4.5’s or Qwen 27B’s; Claude Opus 5.5, Claude Opus 4.8 and GPT-6 Astra fail on 0.8–1.8% of their attempts. The record follows the order in which its experiments ran, so its population shifts late in the stream: in windows 33–38, Qwen3.5-27B supplies 392 of 607 failures, against at most 5% in any earlier window bin.

On held-out τ^2 -bench the user simulator is Claude Opus 4.8 for 320 of the 600 attempts and GPT-6 Astra for 280.

Windows. AppWorld attempts are ordered by start time and cut into 38 windows. A window closes every 50 official failures up to 200, then every 100; the last holds 107. Each window holds a fixed number of failures and between 130 and 3,674 attempts. On the τ^2 -bench training record, each attempt is compared only with attempts that started before it.

Port fidelity. The ACE-style verification playbook is an independent implementation of ACE’s mechanisms [38], written from the paper (arXiv 2510.04618v3) and its public repository, with no pinned upstream commit: itemized bullets with persistent identifiers and helpful and harmful counters, a reflector, a curator that applies incremental deltas, and one Reflector refinement round (ACE’s main runs allow up to five; its cost analysis uses one) without the embedding-based de-duplication of grow-and-refine.

Its prompts are adapted for judging finished attempts rather than copied, and its generator role becomes a judge that returns accept or reject with a success probability. It learns from the same binary outcomes as the library, one reflector call per template per window.

ACE reports agent success, and no run in this program reproduces its offline AppWorld score of 59.4% with DeepSeek-V3.1 and ground-truth labels; the comparison here is between verifiers built on the same record. The precedent judge reads up to 300,000 characters of labeled earlier attempts of the same template.

Validity controls. The held-out τ^2 -bench split was never used to develop either verifier. Both agreement contrasts were preregistered before any held-out judge call, and both were recomputed exactly with an independent implementation. Every panel shown to the peer judge matches the seeded draw that agreement used. Every indexed τ^2 -bench attempt was strictly replayed, and the replayed database agrees with the official database check on all 7,617 attempts that have one.

On AppWorld, each verdict is recorded before its window’s outcomes are read, and the gate reads only windows that have closed. A scan of all 7,398 prompts of the library and its rivals found no grader artifacts, and in the graded run 915 of 915 sampled stored verdicts re-executed with identical results. The library’s difference from its 200-failure copy and the anticipation contrast were preregistered before their runs.

Table 3: **End-state agreement catches more failures than Claude Opus 5.5 judging the same attempts, whether or not the judge sees the four peers, at zero model calls.** Held-out τ^2 -bench: 600 Claude Opus 5.5 attempts at 60 tasks, 68 failures and 532 correct attempts; rates are per attempt. Agreement reads no label; both judges return binary verdicts. ΔJ is against the information-matched peer judge; the registered contrast uses a task-cluster bootstrap stratified by domain.

Verifier	Catch (%) $\uparrow$	False fire (%) $\downarrow$	$J \uparrow$	Calls
Claude Opus 5.5 zero-shot judge	8.8 (6/68)	0.38 (2/532)	0.084	1
Claude Opus 5.5 peer judge, same 4 attempts	30.9 (21/68)	0.19 (1/532)	0.307	1
Agreement, same 4 attempts (registered)	48.5 (33/68)	1.32 (7/532)	0.472	0
Agreement, mean over 50 panels of 4	50.1	1.62	0.484	0
Agreement, 50 panels of 4 attempts by any model	55.8	3.18	0.526	0
ΔJ (points), registered agreement – peer judge			+16.5 [+5.4, +32.1]	

Table 4: **The gap survives every stratum that could explain it away.** Held-out τ^2 -bench, 68 Claude Opus 5.5 failures; catch (%) by stratum. The peer judge saw the database changes of exactly the four attempts that agreement compared. “Plurality” is the most common end state among the other recorded Claude Opus 5.5 attempts at the task.

Question	Failures in the stratum	n	Agreement	Peer judge	Zero-shot
Does the judge see less?	panel held the correct state at least once	54	61.1	38.9	11.1
	correct state was the panel’s strict plurality	30	93.3	50.0	20.0
Were its inputs cut?	no render the judge saw was truncated	68	48.5	30.9	8.8
Easy failures only?	end state reached by no other attempt	4	100.0	25.0	25.0
	repeated, away from the plurality	37	78.4	54.1	13.5
	at the plurality (repeated mistake)	27	0.0	0.0	0.0
One domain?	retail	43	53.5	30.2	9.3
	airline	25	40.0	32.0	8.0

Statistics. Intervals are 95% percentile bootstraps [14]. On τ^2 -bench they resample tasks, stratified by domain on the held-out split and by fold on the training record, with 20,000 draws; on AppWorld they resample templates, with 2,000 draws (20,000 for the agreement-label contrast).

5 End-State Agreement Replaces Judgment with Equality

5.1 Agreement outperforms the information-matched judge

Agreement with the same four attempts catches 33 of the 68 failures. Claude Opus 5.5 judging zero-shot catches 6; shown the same four, it catches 21 (Table 3). Against the information-matched judge, the preregistered difference is $\Delta J = +16.5 [+5.4, +32.1]$: 17.6 more points of catch for 1.1 more points of false fire. Averaged over 50 panels, agreement exceeds the zero-shot judge by a preregistered $+40.0 [+26.0, +61.0]$.

Drawing the four peers from any model raises catch to 55.8%, and J to 0.526, the highest of the preregistered systems in Table 3. The two flags also catch different failures: flagging when either agreement or the peer judge flags catches 52.9% at 1.50% false fire, $\Delta J = +20.74 [+9.49, +36.01]$ over the peer judge alone.

5.2 The judge loses on the comparison, not on information

Four explanations compete with the mechanism. The peer judge might see less than agreement, its inputs might be cut, agreement might take only the easy failures, or the gap might come from where each operating point sits. Table 4 tests each from the logged verdicts.

For 28 failures with a wrong database, the correct end state was the panel’s strict plurality. The judge read those four attempts’ changes, saw the correct state outvote the attempt, and still passed 13; agreement flagged all 28. None of the 30 strict-plurality failures was flagged by the judge alone. No render the judge read was truncated.

Table 5: **The judges catch an attempt that did nothing and miss one that did the wrong thing.** Held-out τ^2 -bench, 68 Claude Opus 5.5 failures classified mechanically by comparing each attempt’s database changes with those of the correct end state at its task; catch (%) by verifier.

Failure class	n	Agreement	Peer judge	Zero-shot
Wrong or missing value in a required write	25	32	8	4
No write at all	22	77	77	18
Some required writes, others omitted	14	29	14	7
Unrequested write only	5	80	0	0
Database correct, reply wrong	2	0	0	0

Neither difficulty nor domain carries the gap. Agreement leads in every stratum of Table 4 where any verifier catches anything, in both domains, and the 27 repeated mistakes defeat all three verifiers alike. Dropping any one of the 60 tasks leaves ΔJ between +14.2 and +19.6.

Both judges return binary verdicts and log no score, so each judge has one operating point. Agreement has a family of them. On the same four attempts, flagging only an attempt that matches none of its peers fires on 1 of 532 correct attempts, the peer judge’s own rate, and catches 22 failures to the judge’s 21.

The registered rule takes 11 more failures for 6 more false fires, a trade the judge cannot make. With more peers the trade improves: scored as the share of 50 seeded panels that flag, agreement catches 41.2% at 1 in 532, at a threshold chosen on the evaluation labels.

Table 5 shows where the two operations part. An attempt that wrote nothing presents an empty change list, and the peer judge flags it as often as agreement does. An attempt that wrote the wrong value, omitted writes among several, or added a write nobody asked for presents a plausible change list. The judges pass nearly all of them; agreement flags those whose state differs from their peers’.

On the 44 failures whose change list looks complete, a wrong value, omitted writes among several, or an unrequested write, agreement catches 16 and the peer judge 4. Of the 15 failures only agreement caught, 6 carried a wrong value and 4 an unrequested write.

Airline task 29 fails by omission. The customer holds VA5SGQ, an insured round trip from Detroit to LaGuardia paid at \$686, and wants nonstop flights to JFK on the same dates. A destination cannot be changed, so the agent books a new reservation, HAT169 out and HAT033 back, on the customer’s Visa. The attempt stopped there. The correct end state also cancels VA5SGQ under its insurance and refunds the \$686.

The zero-shot judge wrote that the new reservation “matches what the customer chose and confirmed”. The peer judge was shown three peers that had cancelled VA5SGQ, two of them beside the same booking, and wrote that the agent “made no unrequested changes”.

The missing cancellation appears nowhere in the attempt’s change list. The attempt’s panel held the correct state twice, a cancellation without a booking once and no change once; its own state matched none of them (Proposition 1, case i).

Retail task 74 fails in a single field. The customer moves a pending laptop order to an i9 with a 256 GB SSD in silver and cancels a second order; the modification lowers the price by \$218.84. The attempt refunded the difference to the PayPal account that paid for the order, a plausible default. Correct attempts refund it to the customer’s credit card.

The zero-shot judge accepted the refund “to the original PayPal account”. Of the 85 recorded Claude Opus 5.5 attempts at the task, the 75 correct ones reach one end state, and the 10 failures scatter over four others, the pattern on which Proposition 1 relies.

In both cases the judges tested the story the end state told against their reading of the policy, and a complete, plausible story passed. Agreement tested equality with each peer, and a wrong state fails that test however well it reads.

5.3 Agreement sharpens as attempts accumulate

The preregistered panel size is not a lucky choice. On held-out τ^2 -bench, agreement’s mean catch over 50 panels lies between 44.1% and 58.3% for every k from 1 to 9, and its J between 0.426 and 0.545, above the information-matched

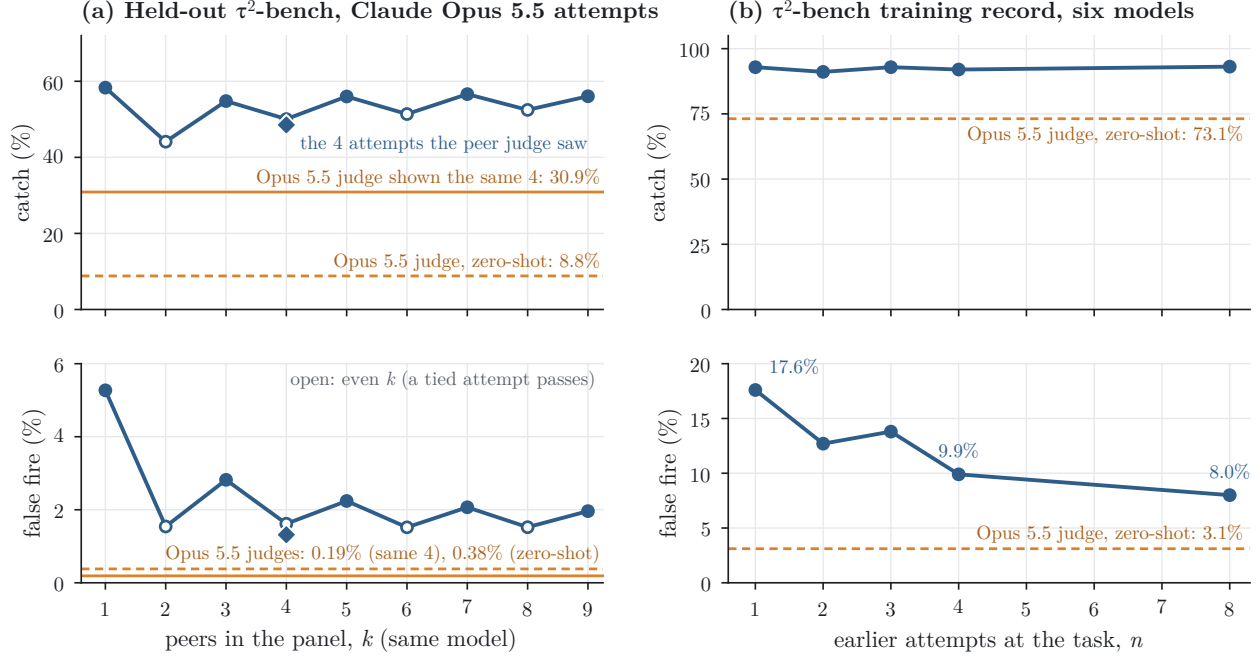


Figure 3: **Agreement out-catches both Claude Opus 5.5 judges at every panel size, and its false fire falls as peers accumulate, at zero model calls per check.** (a) Held-out τ^2 -bench, 600 Claude Opus 5.5 attempts (68 failures): catch and false fire per attempt against the number k of same-model peer attempts, mean of 50 seeded panels; the diamond is the preregistered panel of four that the peer judge also saw. Open markers are even k , where an attempt tied for the most common state passes. Lines mark the two judges. (b) τ^2 -bench training record, 1,133 attempts at 104 tasks by six actor models, replayed in deployment order: catch and false fire against the number n of earlier attempts at the task, with the Claude Opus 5.5 zero-shot judge on the same items.

judge’s 0.307 at every k (Figure 3a). Even panels catch and fire less than odd ones, because an even panel admits more ties and a tied attempt passes.

On the training record, each attempt is compared only with attempts that started before it. False fire falls from 17.6% with one earlier attempt to 9.9% with four and 8.0% with eight, while catch stays near 93% (Figure 3b).

The mechanism is Proposition 1(iii): a correct attempt is flagged only when a wrong state outnumbers the correct one among its predecessors, and each added attempt makes that rarer. Failures are the majority of this record, 618 of 1,133 attempts, and agreement still separates them.

With four other attempts drawn from the whole record, agreement catches 92.1% at 6.5% false fire; the zero-shot judge catches 73.1% at 3.1%, a difference of $\Delta J = +15.6 [+10.7, +20.8]$. A harness that uses the judge until three earlier attempts exist, and agreement afterward, catches 93.7% at 5.6%: $\Delta J = +18.0 [+12.3, +23.8]$ over the judge. Each added peer costs a hash comparison; each judge call costs a model call.

5.4 Repeated mistakes, and the panels that reach them

The 27 failures at Claude Opus 5.5’s most common end state are repeated mistakes, the case of Proposition 1(ii); two of them fail on communication with a correct database, the case of (iv). Panels of the same model hold such a mistake whenever the model makes it often. Panels drawn from Claude Haiku 4.5 catch 71.1% of the 27, because a panel of another model holds Claude Opus 5.5’s error only when that model makes the same one.

Diversity of harness works the same way: Claude Opus 5.5 panels drawn from any run catch 51.6% of the 66 wrong-database failures, while panels limited to other trials of the same run, which average 2.3 peers, catch 28.4% of the 60 that have such peers.

Repeated mistakes mark the boundary between two uses of agreement. As a certificate, consensus fails: VeriHarness reports that the benchmark’s grades mark 34% of the consensus values in Claude Opus 4.8’s ten-rollout APEX-Agents

Table 6: **Revision history of one compiled check.** The template that asks the agent to follow every artist of one genre’s songs in the user’s playlists, in the pass/fail run: 314 attempts at three instances, 11 failures and 303 correct attempts. At each gate the candidate also fired on none of 150 successes of other templates. No version fired on a correct attempt.

Window	Event	Check and gate
1	Anticipation arm writes a check from two correct Claude Opus 5.5 attempts, before any failure	Missing, removed and extra follows; silent on 6 past successes
4	First failure: a Claude Haiku 4.5 attempt leaves one reggae artist unfollowed. The failure-only library has no check and passes it; the anticipation check flags it	Minted from that failure: the same three conditions; silent on 23 past successes, catches 1 of 1 past failure
6–7	Three Claude Haiku 4.5 failures, each with artists unfollowed	All three caught
8	A Claude Haiku 4.5 attempt follows all five classical artists, then submits <code>answer=5</code> to an action task; passed	Matured: wider instruction parser, and no answer on an action task; silent on 93 past successes, catches 5 of 5 past failures (old check 4)
23–37	Six failures: five by Qwen 27B, one by Claude Haiku 4.5; three Qwen 27B attempts submit answers	All six caught; the copy frozen after 200 failures passes the three answers

pools incorrect [36], and False Frontiers finds that “majority stability does not imply correctness when labelers share a model and evidence” [5].

As a flag, disagreement is precise. An attempt that leaves the plurality is wrong in 33 of the 40 cases agreement flags on held-out τ^2 -bench, at a frozen policy that no agreement signal trained.

6 Compiled Checks Accrue, Transfer to Stronger Models, and Depreciate

Runs. In the *pass/fail run*, the writer’s prompts keep the instruction, the trajectory, an end-state change summary and the binary outcome; every grader-derived field is removed. A separate *graded run* also shows the writer the grader’s report on past failures.

Both include an *anticipation* arm, which also writes a check from up to two correct attempts of a template before its first failure. The graded run adds a *no-Opus* arm, which learns only from Claude Haiku 4.5, Qwen 27B and GPT-6 Astra attempts and checks every attempt. Frozen copies stop learning after 50, 200, 800 or 1,800 failures.

Rivals. Two learning rivals use Claude Opus 5.5 and the same feedback: the precedent judge, which reads labeled earlier attempts of the same template, and an ACE-style verification playbook with a Playbook, Reflector and Curator [38]. Both return a success probability p from 0 to 100 and flag at $p \leq \tau$. Their thresholds were matched to the library’s false fire on the evaluation labels, which favors them.

Both read a rendered text view of each attempt’s state changes and relevant records, while a check queries the full start and end databases. Database access is part of what a check compiles: it reads, at no token cost, records that a judge would have to hold in its context.

6.1 Anatomy of a compiled check

Table 6 follows one check through the pass/fail run, and Appendix B.5 reproduces its final version. The check computes the requirement before it reads the result. It parses the genre from the instruction, collects the artists of each song of that genre in the user’s playlists in the start state, and compares that set with the artists the user follows in the end state. It flags three distinct errors, each with the ids it found: an expected artist not followed, a previously followed artist removed, and a newly followed artist outside the expected set.

The template first failed in window 4, when a Claude Haiku 4.5 attempt left artist 30 of a reggae playlist unfollowed. The failure-only library had no check and passed it. Claude Opus 5.5 minted one from that failure, and the gate admitted it after it stayed silent on 23 past successes. The check then caught three Claude Haiku 4.5 failures in windows 6 and 7.

In window 8 a Claude Haiku 4.5 attempt followed all five classical artists correctly and then completed the task with `answer=5`. The task requires no answer, and the check passed the attempt.

Table 7: **Compiled checks catch 34.6 points more never-seen failures than an ACE-style verification playbook at matched false fire, fire a third as often as its native verdict, and use a twentieth of the tokens.** AppWorld verifiers learned from past official pass/fail outcomes without grader text; Claude Opus 5.5 throughout. Catch is over 1,036 never-seen failure states in windows 5–38; false fire is over 627 never-seen correct states in those windows. Token totals cover the whole stream, and the library’s covers two arms. The matched threshold uses evaluation labels.

Verifier	Catch (%) $\uparrow$	False fire (never-seen correct) $\downarrow$	Tokens $\downarrow$
Compiled checks, learning	90.1	7 (1.1%)	5.7M
Compiled checks, frozen after 200 failures	66.0	21 (3.3%)	–
ACE-style verification playbook, $p \leq 7$ (false fire matched)	55.5	6 (1.0%)	117M
ACE-style verification playbook, native verdict	87.2	20 (3.2%)	117M

The writer saw that the attempt had failed, with its trajectory and end state beside two correct attempts, and no grader report. From those it inferred the error: it added a fourth condition, that an action task must submit no answer, and widened the instruction parser. The revision caught all 5 of the template’s past failures, where the old check caught 4.

After the population changed, the check caught all 6 of the template’s later failures, three of them Qwen 27B attempts that followed the right artists and then submitted answers: 5.0, “All classical artists have been followed.”, and “All classical artists from your playlists are already followed.” A check written from a Claude Haiku 4.5 error caught Qwen 27B errors of the same kind. The copy frozen after 200 failures holds the window-4 version and passes all three answers.

6.2 Compiled checks outperform an ACE-style verification playbook on a twentieth of the tokens

After its first 200 failures, the pass/fail library catches 90.1% of never-seen failure states in windows 5–38 and flags 1.1% of never-seen correct states (Table 7). Per attempt, it catches 96.4% of failures and fires on 0.026% of correct attempts. Four of its 7 false fires came from one check in window 5, revoked there; the other 3 fell in windows 33–35, where the population changed.

At matched false fire, the ACE-style verification playbook catches 55.5%. At its native verdict it catches 87.2% at 3.2% false fire, 2.9 times the library’s rate. The population shift separates them further: at matched false fire, the playbook catches 30.9% of the final bin’s never-seen failure states, where the library catches 87.6%.

The playbook ran as designed. All 4,518 of its calls completed without a refusal or error, and the playbook grew from 256 to 623 bullets through 913 applied deltas.

Its misses resemble the judges’ misses on τ^2 -bench. One template asks the agent to follow the artists of the songs and albums the user has liked on Spotify. In window 23 a Claude Haiku 4.5 attempt followed 18 artists where the task required 20. The playbook’s judge, reading 108 of its 487 bullets, 6 of them for this template, accepted the attempt at a success probability of 80: “Clean, deduplicated insert-only follow set of 18 artists ... with no extraneous changes”.

The check, minted in window 1 from one failure, flagged it: “2 artist(s) of liked songs and albums are not followed at the end: artist ids [32, 36] (expected 20 artists followed, found 18).” Across windows 5–38 the playbook passed 390 of the library’s 933 catches at its matched threshold, and accepted 113 of them even at its native verdict.

The cost difference is structural. A check is a Python function that runs without a model call. The playbook’s lessons are text that Claude Opus 5.5 must reapply at each attempt, and a precedent judgment is a Claude Opus 5.5 call per attempt. The library’s two arms used 5.7M tokens over the whole stream in 261 writer calls; the ACE playbook used 117M in 4,518 calls, and the precedent judge 123M on its evaluation items alone (Appendix A.5).

Construction cost amortizes because the stock stops growing. In windows 11–38, with a check held for every template, the library made 52 writer calls while checking 22,438 attempts, one call per 432 checked attempts. Over the same windows the playbook made 2,896 calls and the precedent judge 1,667, on 1,691 judged items.

The strongest learning rival is the precedent judge. Over the preregistered late windows 29–38, at its false-fire-matched threshold $p \leq 5$, the library catches 89.9% of never-seen failure states and the judge 82.6%, a difference of +7.3 [+1.5, +13.4], on under a twentieth of the judge’s tokens.

A per-attempt rate as low as 0.026% must be read against a novelty flag, which fires on every end state not previously seen correct. A novelty flag catches every failure at 2.35% per-attempt false fire, but on never-seen states it flags

Table 8: **The library stops growing in window 8; its value keeps changing.** Pass/fail run. Left: accrual of the learning library by window bin, with failures (per attempt) the library missed, checks held at the end of the bin, and admitted mints, maturations and other revisions (re-mints, retries and repairs). Right: catch (%) of the bin’s never-seen failure states by the learning library and by copies frozen after 50, 200, 800 and 1,800 failures; a dash marks a bin before the freeze.

Windows	Accrual						Catch of never-seen failure states (%)					
	Failures	Missed	Checks	Mints	Matur.	Other	States	Learning	50	200	800	1,800
1–4	200	168	39	30	5	9	150	7.3	–	–	–	–
5–10	600	60	48	5	29	6	277	83.4	22.0	70.8	–	–
11–18	800	9	48	0	6	2	171	94.7	10.5	70.2	88.3	–
19–26	800	8	48	0	7	0	181	95.6	15.5	74.6	87.8	–
27–32	600	8	48	0	7	0	148	94.6	5.4	79.7	84.5	87.8
33–38	607	37	48	0	23	4	259	87.6	10.0	44.4	68.0	68.7

everything and has $J = 0$. The library separates never-seen failure states from never-seen correct ones, and it fires on correct attempts at one ninetieth of the novelty flag’s rate.

6.3 Frozen checks depreciate through forgone revision

Frozen copies depreciate at every freeze point (Table 8; Figure 4). The copy frozen after 200 failures catches a preregistered 24.0 [+16.4, +32.8] points less than the learning library on the later never-seen failure states. Copies frozen after 800 and 1,800 failures catch 12.0 [+5.9, +18.0] and 10.6 [+4.3, +19.2] points less, and a copy frozen after 50 failures 76.5 [+65.4, +84.8] less.

The size of the library does not explain the curve. From window 8 the library held one check per template and added none, so the copies frozen after 800 and 1,800 failures hold a check for every template, as the learning library does. Their depreciation is forgone revision alone (Definition 3).

After window 10 the learning library admitted 43 maturations and 3 repairs; each answered a failure the old check had missed or a correct attempt it had flagged, and by Proposition 2 each kept what its predecessor caught on the gate’s samples.

Each miss is material for a revision. Of the library’s 103 misses in windows 5–38, 66 came from a check that lacked a condition, 18 from a check that judged itself not applicable, 15 from templates with no check yet and 4 from a check that raised an error. In 95 of the 103 the miss triggered a request to write or revise the template’s check, and for 88 of them the gate admitted the result. A frozen copy receives none of them.

Revisions broaden a check beyond the failures that prompted them, as the wider parser of Section 6.1 did. On the never-seen failure states of templates the 200-failure copy already covered, the learning library catches 17.6 [+11.1, +24.4] points more. On the 171 first appearances of a failure mode, where a mode is a template paired with the set of grader requirements it violated, it catches 67.8% [59.0, 74.9] against 54.4%.

Depreciation is steepest when the population changes. In windows 33–38 the learning library answered Qwen 27B’s arrival with 23 admitted maturations and 3 repairs, its largest burst of revision since windows 5–10. It caught 87.6% of the bin’s never-seen failure states, including 86.3% of Qwen 27B’s, while the copies frozen after 200 or more failures fell to between 44.4% and 68.7%. Learning also lowers false fire: the 200-failure copy flags 3.3% of never-seen correct states, three times the learning library’s rate.

6.4 Checks compiled from other models’ failures verify Claude Opus 5.5

Claude Opus 5.5 fails on only 0.8% of its AppWorld attempts, and Claude Opus 4.8 on 1.8%. A verifier for Claude Opus 5.5 must therefore draw on other models’ failures. The no-Opus arm of the graded run learns from no Claude Opus 5.5 or Claude Opus 4.8 attempt; Claude Opus 5.5 still writes its checks, from Claude Haiku 4.5, Qwen 27B and GPT-6 Astra failures and their grader reports. Of the 3,407 failures available to it, all but 20 came from Claude Haiku 4.5 or Qwen 27B, both of which fail far more often than Claude Opus 5.5 on this record.

On the same 144 instances, the no-Opus checks caught 30 of the 32 never-seen Claude Opus 5.5 and Claude Opus 4.8 failure states, and 57 of their 59 failure attempts. They fired on none of the 7,185 correct attempts by those two models.

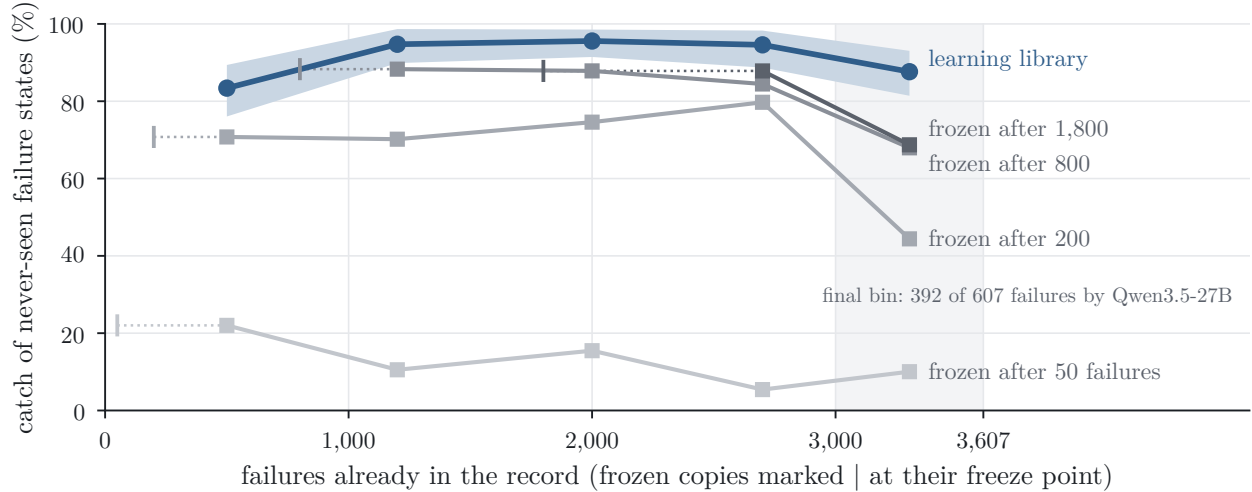


Figure 4: **Frozen copies fall behind at every freeze point.** Catch of never-seen AppWorld failure states by window bin. The library learns from past official pass/fail outcomes without grader text; its band is a 95% template-cluster bootstrap interval. Vertical marks indicate when copies were frozen, and dotted lines connect each freeze to the first scored bin. The shaded final bin has 392 Qwen 27B failures out of 607; there, catch falls to 44.4–68.7% for copies frozen after 200 or more failures, while the learning library holds 87.6%.

Table 9: **Frozen checks catch two thirds of failure states on instances never used to write or admit them, with no false fire.** AppWorld, 46 templates; checks written by Claude Opus 5.5 from attempts at writing instances and frozen before evaluation. Units differ by row and are stated.

Evaluation split	Failures	Catch (%)	Correct	False fires
Writing instances, held-out runs, no exact repeat of a writing attempt	117 attempts	82.1	80 attempts	0
Other instances, other runs	367 states	66.2	370 states	0

By Proposition 3, a check’s verdict depends on the instruction, the start state, the end state and the answer, so it cannot see which actor produced a state. It flags a Claude Opus 5.5 failure when that failure violates the requirement the check re-derives, whoever’s failure first prompted the check.

6.5 Checks hold on instances they were never written from

The never-seen states above fall on the same 144 instances, so they could reflect learning tied to an instance. A separate study, run before the library, separates the two. Claude Opus 5.5 wrote one check per template for 46 templates from attempts at some instances, a gate on those instances admitted them, and the checks were frozen before any held-out result was read (Table 9).

On other instances from other runs, the frozen checks caught 66.2% of 367 distinct failure states and fired on none of 370 distinct correct states. Each check re-derives the requirement from the new instance’s instruction and start state, so the requirement moves with the instance.

Several misses come from checks that parse only the phrasing of the instance they were written from, and maturation repairs that fault: on a fresh slice, it raised catch on 166 never-seen failure states from 41.0% to 63.9%, +22.9 [+13.6, +34.3], with 1 of 52 never-seen correct states flagged.

6.6 Checks seeded from correct attempts anticipate first failures

A failure-only library cannot catch a template’s first failure, because it has no check until a failure has been missed. A correct attempt carries what a check needs: the end state the template requires.

In the graded run, on 110 instance first failures, the anticipation arm raises catch from the failure-only library’s 20.0% to 75.5%, a preregistered +55.5 [+44.6, +65.6], and it fired on none of 7,915 correct attempts in windows 2–12. On

Table 10: **Revision, seeding and official outcomes carry the library; grader text adds little; diverse panels help agreement.** Each row changes one component and reports the measured contrast. AppWorld rows are catch (%) of never-seen failure states in windows 5–38 unless stated, with 95% template-cluster intervals; the τ^2 -bench row is J per attempt.

Change	With	Changed	Δ [95% CI]
<i>Compiled checks, AppWorld pass/fail run</i>			
Stop revising after 200 failures (preregistered)	90.1	66.0	−24.0 [−32.8, −16.4]
Stop revising after 1,800 failures, with every check held (568 states)	–	–	−10.6 [−19.2, −4.3]
Remove seeding from correct attempts (113 instance first failures)	74.3	20.4	−54.0 [−64.7, −42.6]
Show the writer the grader’s report on past failures	90.1	92.5	+2.4 [+0.8, +4.4]
Drop 30% of official outcomes at random	90.1	87.5	−2.6 [−4.3, −1.1]
Replace official outcomes by the agents’ completion flags	90.1	5.1	−84.9 [−88.5, −80.9]
<i>End-state agreement, τ^2-bench training record (1,133 attempts, six actors)</i>			
Draw the four peers from the same actor model instead of other models	0.832	0.778	−5.5 [−8.1, −2.8]

the 44 instances whose own correct attempt the writer was not shown, catch rises from 29.5% to 65.9%, because a seeded check encodes the template’s requirement, which holds for instances absent from its seed attempts.

The pass/fail run shows the same lift without grader text, 74.3% against 20.4% on its 113 instance first failures (Table 10).

6.7 What each component contributes

Table 10 isolates each part. Without further learning the library misses a third of the never-seen failures, and without seeding it misses four in five first failures. Official outcomes carry it too, but they need not be complete: losing 30% of them costs 2.6 points, while replacing them by the agents’ own completion flags costs 84.9. For agreement, peers drawn from other models raise J over peers of the attempt’s own model.

7 Labels: Label-Free Operating Points Hold, Self-Reports Poison Learning

A learning verifier needs outcomes, and a deployed harness gets them noisy. Consider a verifier that flags when either the library or the precedent judge flags, both learning from labels in which each outcome was flipped with probability 0.10. At the judge’s native verdict, an operating point that takes no threshold from labels, it keeps 98.4% of its clean catch, at 3.78% false fire on the 687 never-seen correct states of windows 2–38.

The same verifier with its threshold tuned on the noisy labels keeps 72.8%. At 30% flips the native point keeps 92.2%, at 6.55% on the same 687.

The agents’ own completion flags are the cheapest labels, and the worst. On the AppWorld record they marked 3,309 of 3,607 failures as done. Fed completion flags, the library catches 5.1% of never-seen failure states, against 90.1% from official outcomes. Only 298 failures arrive as failures to learn from, and the rest enter the gate’s success set, where a check that flags them breaks negative selection.

Agreement supplies better labels. Labeling an attempt a success when it agrees with the consensus of other attempts never called a failure a success on this record, and withheld the success label from 7.5% of successes. A prequential precedent judge fed these one-sided labels, which report agreement as success and leave the rest unlabeled, reaches $J = 0.880$ on never-seen states: 96.3% catch at 8.3% false fire. Completion flags give $J = 0.240$ and official labels $J = 0.939$, and the difference between agreement labels and completion flags is +64.1 [+52.9, +70.3].

The labels must be one-sided. Two-sided labels, which call every non-plurality state a failure, push the judge’s false fire to 56.5%, because a correct variant is a minority when it first appears. The one-sided label is positive-unlabeled learning [15].

False Frontiers shows what happens when agreement becomes the reward of an optimizing loop: proposer and solver “increasingly agree on shared errors” [5]. Here no policy is trained on agreement labels; they feed a verifier, scored against official outcomes.

8 Related Work

From self-consistency to end-state agreement. Self-consistency samples several reasoning paths and takes a majority vote over their answers [33]. MBR-exec scores each sampled program by its agreement with the others’ execution results, without ground-truth output [28]; AlphaCode groups programs that produce the same outputs on generated inputs [22]; CodeT combines tests written by the same model with agreement among code samples [3]. Universal Self-Consistency spends one more model call to pick the most consistent response [6].

Each selects an answer. End-state agreement is the immediate descendant of the premise AlphaCode and CodeT state, that correct solutions behave alike while incorrect ones are diverse [3, 22], carried from program outputs to the database a multi-turn tool agent leaves, and used to reject an attempt rather than select one.

The nearest contemporary use of agreement is VeriHarness, which treats disagreement within a pool of rollouts as a pointer to claims worth checking and has the same model resolve them against workspace evidence [36]. It reports majority voting at +0.8 points for Claude Opus 4.8, against +4.1 for its verifier’s selection alone and +6.4 with revision.

The difference from this paper is the object compared. VeriHarness compares claims across rollouts and has a model test them; end-state agreement votes over environment states, and its verdict is the count itself, with no model resolving it.

SelfCheckGPT and semantic entropy also read disagreement as an error signal, over sampled text [16, 25].

Correlated failure. N-version programming and pseudo-oracles treat independent implementations as an oracle [1, 4, 9], and independently written versions fail together more often than independence predicts [13, 21]. Language models do too, the more so the stronger they are [18, 20]. False Frontiers shows the extreme case under optimization: when agreement is the reward, proposer and solver “increasingly agree on shared errors” [5], and majority-agreement rewards in unsupervised reinforcement learning can rise while accuracy falls [12].

Claude Opus 5.5 is the hardest actor in this paper for agreement for the same reason, and the repeated mistakes of Section 5.4 are where its coincident failures appear.

Judges, critics and verifier training. Trained verifiers and process reward models score solutions [7, 23, 32]; generative verifiers cast verification as next-token prediction [37]; critics and tool-interactive critique add feedback the model lacks [19, 26]. Beyond the Verdict trains ARBITER, a 35B agentic verifier, with reinforcement learning to back its verdicts with counterexamples [11].

Rethinking Verification in Agent Harnesses compares 17 verification designs and finds that an independent critic with access to the live sandbox offers a more favorable accuracy–cost tradeoff than completion gates [39]; it scores the critic through post-revision accuracy, and one review averages 38.2k critic tokens. Each of these pays a model call at every check. On τ^2 -bench, the comparison here holds the information fixed and changes the operation, from a model’s inference to an equality.

Generated tests, hardening and playbooks. HarnessPro scores candidate harnesses with ten executable tests that a model writes once per task, from the task and an initial trajectory, and keeps fixed “throughout the self-improvement process” [27]. Compiled checks are also model-written and run without a model call, but they are written per template from past pass/fail outcomes and then keep learning: each missed failure can revise a check, and Section 6.3 measures what a fixed copy, the analogue of fixed generated tests, loses.

Weak-to-Strong Verifier Hardening patches programmatic verifiers against reward hacking with a weaker hacker and a fixer, and admits a repair only if, among other gates, every known legitimate solution of the task, reference solutions included, still passes on replay [35]. The admission gate here applies the same principle, negative selection [17], to model-written checks of non-adversarial failures: a check must stay silent on the sampled past successes of its template.

ACE evolves a playbook of bullets through a Generator, a Reflector and a Curator to guide the agent that acts [38]; the ACE-style verification playbook of Section 6.2 is this paper’s adaptation of that loop to judging finished attempts. Reflexion, Self-Refine, ExpeL and Dynamic Cheatsheet likewise change how an agent acts on its next task [24, 29, 30, 40]. A compiled check leaves the agent as it is, and its value is measured window by window as a prequential forecast [10].

Program paper. This paper is one of five companion papers to Cumes [8], which reports an experiment on a self-governing research organization of frontier language-model agents and summarizes this paper’s end-state agreement and compiled-check results.

9 Conclusion

Both zero-call verifiers replace a judgment with a comparison. End-state agreement compares an end state with the end states of other attempts at the same task; a compiled check compares it with a requirement re-derived from the instruction and the start state. A judge must reconstruct that requirement, read the change and decide, and a plausible change list passes: Claude Opus 5.5 judging zero-shot accepted a new booking that left the original reservation active.

The comparison changes the cost structure of verification. Agreement’s false fire falls as attempts accumulate, from 17.6% with one earlier attempt to 8.0% with eight on the training record, because correct attempts coincide while failures scatter, and each added peer costs a hash comparison. A check costs one model call to write and none to run. Because it cannot see which model produced a state, checks compiled from Claude Haiku 4.5 and Qwen 27B failures verify Claude Opus 5.5.

The same property sets the limit of a frozen library. A check encodes the failures the record had produced when it was last revised. With the library fixed at 48 checks, coverage tracks the graded outcomes that keep reaching it, and a copy frozen with every check in place loses 10.6 points. Verification in a harness that runs unattended is an asset that depreciates unless it keeps learning from outcomes.

A done-check for a frontier tool agent must be a comparison, and it must keep learning. The next measurement places agreement flags and check verdicts, each with its evidence sentence, in front of the acting agent before its completion call, and counts the failures they turn into repairs.

Acknowledgments

We thank the AppWorld and τ^2 -bench maintainers and the authors of the concurrent work for their open releases.

References

- [1] A. Avizienis. The N-Version Approach to Fault-Tolerant Software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, 1985. doi: 10.1109/TSE.1985.231893.
- [2] 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.
- [3] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. CodeT: Code generation with generated tests. In *International Conference on Learning Representations (ICLR)*, 2023. arXiv:2207.10397.
- [4] Liming Chen and Algirdas Avizienis. N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation. In *Proceedings of the 8th International Symposium on Fault-Tolerant Computing (FTCS-8)*, 1978. doi: 10.1109/FTCSH.1995.532621. Reprinted in Twenty-Fifth International Symposium on Fault-Tolerant Computing, Highlights from Twenty-Five Years, 1995, p. 113.
- [5] Meijia Chen, Hao Li, Zheng Lu, Hongshan Lin, Junbai Tian, Yichen Liu, Zijun Tian, Yufan Zou, Shuhan Sun, Hanxin Chen, Zeyu Zhang, Weizhi Du, Yueting Li, Tianyu Shi, and Alaa Khamis. False Frontiers: Diagnosing and Mitigating Co-Cheating in Self-Evolving Search Agents. *arXiv preprint arXiv:2609.39102*, 2026.
- [6] Xinyun Chen, Renat Aksitov, Uri Alon, Jie Ren, Kefan Xiao, Pengcheng Yin, Sushant Prakash, Charles Sutton, Xuezhi Wang, and Denny Zhou. Universal Self-Consistency for Large Language Model Generation. *arXiv preprint arXiv:2311.17311*, 2023.
- [7] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training Verifiers to Solve Math Word Problems. *arXiv preprint arXiv:2110.14168*, 2021.

- [8] Lindell S. Cumes. Evidence-Gated Recursion: An Experiment on a Self-Governing Research Organization of Frontier Language-Model Agents. QWNTL Labs; alphaXiv 2610.nenya-headroom-law-recursive-harnesses (v2), <https://qwntl.com/frontier>, 2026.
- [9] Martin D. Davis and Elaine J. Weyuker. Pseudo-oracles for non-testable programs. In *Proceedings of the ACM '81 Conference*, pages 254–257, 1981. doi: 10.1145/800175.809889.
- [10] A. P. Dawid. Present position and potential developments: Some personal views. statistical theory: The prequential approach. *Journal of the Royal Statistical Society. Series A (General)*, 147(2):278–292, 1984. doi: 10.2307/2981683.
- [11] Zheye Deng, Sha Li, Changlong Yu, Xin Liu, Qin Lu, Priyanka Nigam, and Yangqiu Song. Beyond the Verdict: Agentic Code Judging with Verifiable Feedback. *arXiv preprint arXiv:2610.18760*, 2026.
- [12] Peize Ding. Collective Dynamics of Self-Reinforcement: How Cross-Task Coupling Drives Policy Collapse in Unsupervised RLVR. *arXiv preprint arXiv:2610.37553*, 2026.
- [13] Dave E. Eckhardt and Larry D. Lee. A theoretical basis for the analysis of multiversion software subject to coincident errors. *IEEE Transactions on Software Engineering*, SE-11(12):1511–1517, 1985. doi: 10.1109/TSE.1985.231895.
- [14] B. Efron. Bootstrap Methods: Another Look at the Jackknife. *The Annals of Statistics*, 7(1):1–26, 1979. doi: 10.1214/aos/1176344552.
- [15] Charles Elkan and Keith Noto. Learning Classifiers from Only Positive and Unlabeled Data. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 213–220, 2008. doi: 10.1145/1401890.1401920.
- [16] Sebastian Farquhar, Jannik Kossen, Lorenz Kuhn, and Yarin Gal. Detecting hallucinations in large language models using semantic entropy. *Nature*, 630(8017):625–630, 2024. doi: 10.1038/s41586-024-07421-0.
- [17] S. Forrest, A.S. Perelson, L. Allen, and R. Cherukuri. Self-nonself discrimination in a computer. In *Proceedings of 1994 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 202–212, 1994. doi: 10.1109/RISP.1994.296580.
- [18] Shashwat Goel, Joschka Struber, Ilze Amanda Auzina, Karuna K. Chandra, Ponnuram Kumaraguru, Douwe Kiela, Ameya Prabhu, Matthias Bethge, and Jonas Geiping. Great models think alike and this undermines AI oversight. *arXiv preprint arXiv:2502.04313*, 2025.
- [19] Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. CRITIC: Large language models can self-correct with tool-interactive critiquing. In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2305.11738.
- [20] Elliot Kim, Avi Garg, Kenny Peng, and Nikhil Garg. Correlated errors in large language models. In *International Conference on Machine Learning (ICML)*, 2025. arXiv:2506.07962.
- [21] John C. Knight and Nancy G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering*, SE-12(1):96–109, 1986. doi: 10.1109/TSE.1986.6312924.
- [22] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d’Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022. arXiv:2203.07814.
- [23] Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let’s Verify Step by Step. *arXiv preprint arXiv:2305.20050*, 2023.

- [24] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegrefe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-Refine: Iterative Refinement with Self-Feedback. *arXiv preprint arXiv:2303.17651*, 2023.
- [25] Potsawee Manakul, Adian Liusie, and Mark J. F. Gales. SelfCheckGPT: Zero-Resource Black-Box Hallucination Detection for Generative Large Language Models. *arXiv preprint arXiv:2303.08896*, 2023. EMNLP 2023.
- [26] Nat McAleese, Rai Michael Pokorny, Juan Felipe Ceron Uribe, Evgenia Nitishinskaya, Maja Trebacz, and Jan Leike. LLM critics help catch LLM bugs. *arXiv preprint arXiv:2407.00215*, 2024.
- [27] Zhuoshi Pan, ZhiYuan Feng, Yukuo Cen, Honglin Lin, Junru Lu, Zhongxia Chen, Shuo Yang, H. Vicky Zhao, and Jie Tang. HarnessPro: Recursive Harness Self-Improvement with Pareto Optimization for Performance-Cost Trade-offs. *arXiv preprint arXiv:2610.26770*, 2026.
- [28] Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I. Wang. Natural Language to Code Translation with Execution. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2022.
- [29] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language Agents with Verbal Reinforcement Learning. *arXiv preprint arXiv:2303.11366*, 2023.
- [30] Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic Cheatsheet: Test-Time Learning with Adaptive Memory. *arXiv preprint arXiv:2504.07952*, 2025.
- [31] 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. *arXiv preprint arXiv:2407.18901*, 2024. ACL 2024.
- [32] Jonathan Uesato, Nate Kushman, Ramana Kumar, Francis Song, Noah Siegel, Lisa Wang, Antonia Creswell, Geoffrey Irving, and Irina Higgins. Solving math word problems with process- and outcome-based feedback. *arXiv preprint arXiv:2211.14275*, 2022.
- [33] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-Consistency Improves Chain of Thought Reasoning in Language Models. *arXiv preprint arXiv:2203.11171*, 2022. ICLR 2023.
- [34] W. J. Youden. Index for rating diagnostic tests. *Cancer*, 3(1):32–35, 1950. doi: 10.1002/1097-0142(1950)3:1<32::AID-CNCR2820030106>3.0.CO;2-3.
- [35] Chong Zeng, Ziqian Zhong, Kexun Zhang, Ivan Bercovich, Ivgeni Segal, Yuzhen Mao, Shashwat Saxena, and Aditi Raghunathan. Scalable Weak-to-Strong Verifier Hardening for Agent Environments. *arXiv preprint arXiv:2610.15924*, 2026.
- [36] Caiqi Zhang, Rujun Han, Zifeng Wang, Zoey CuiZhu, Nigel Collier, Tomas Pfister, and Chen-Yu Lee. VeriHarness: Scaling Agentic Verification for Long-Horizon Tasks. *arXiv preprint arXiv:2610.00972*, 2026.
- [37] Lunjun Zhang, Arian Hosseini, Hritik Bansal, Mehran Kazemi, Aviral Kumar, and Rishabh Agarwal. Generative verifiers: Reward modeling as next-token prediction. In *International Conference on Learning Representations (ICLR)*, 2025. arXiv:2408.15240.
- [38] Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic Context Engineering: Evolving Contexts for Self-Improving Language Models. *arXiv preprint arXiv:2510.04618*, 2025. ICLR 2026.
- [39] Zhengxin Zhang, Erkang Zhu, Yin Lin, Claire Cardie, Bolin Ding, and Jingren Zhou. Rethinking Verification in Agent Harnesses. *arXiv preprint arXiv:2610.34577*, 2026.

- [40] Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM Agents Are Experiential Learners. *arXiv preprint arXiv:2308.10144*, 2023. AAAI 2024.
- [41] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. *arXiv preprint arXiv:2306.05685*, 2023. NeurIPS 2023 Datasets and Benchmarks Track.
- [42] Mingchen Zhuge, Changsheng Zhao, Dylan Ashley, Wenyi Wang, Dmitrii Khizbullin, Yunyang Xiong, Zechun Liu, Ernie Chang, Raghuraman Krishnamoorthi, Yuandong Tian, Yangyang Shi, Vikas Chandra, and Jürgen Schmidhuber. Agent-as-a-judge: Evaluate agents with agents. *arXiv preprint arXiv:2410.10934*, 2024.

A Experimental Details

A.1 Held-out τ^2 -bench index

The held-out split holds 40 retail and 20 airline tasks. A frozen index holds 7,641 graded attempts by five models: Claude Opus 5.5 (4,403), Claude Haiku 4.5 (2,584), GPT-6 Astra (404), Gemini 3 Flash (239) and Qwen3.5-397B (11). Before judging, 468 of 8,109 recorded attempts were excluded as invalid, aborted, flagged, affected by infrastructure, or lacking an outcome. Each indexed attempt was strictly replayed, and the replayed database agrees with the official database check on all 7,617 attempts that have one.

A.2 Self-verification measurements

The motivation figures of Section 1 come from three measurements on AppWorld, each with its own setup.

Self-acceptance. Claude Opus 5.5 worked 417 test-challenge tasks three times each, with a Claude Opus 5.5 self-verifier that could roll the attempt back; pass¹ was 93.5 and pass³ 92.1. Twenty-two tasks failed in all three repeats. On those 66 failed attempts the self-verifier marked 65 as verified. Over every failed attempt in the three repeats it accepted 78 of 81.

A fresh judge. A preregistered follow-up showed a fresh Claude Opus 5.5 judge only the task instruction, the initial state (up to 8,000 characters), the submitted answer and the attempt’s database changes (up to 12,000 characters), with no code, outputs, reasoning or status. It accepted 64 of the 66 systematic failures and rejected 5 of 157 passing attempts.

Re-checking. A preregistered study sampled a Claude Haiku 4.5 outcome check (temperature 1.0) three times on each of 69 AppWorld train and dev trajectories whose actor had claimed completion; 39 were officially wrong. A beta-binomial fit to the per-item false-acceptance rates gives $\alpha = 0.379$ and $\beta = 0.211$, an intraclass correlation of $1/(\alpha + \beta + 1) = 0.63$. The fit reproduces the observed false acceptance after one, two and three samples (0.641, 0.556 and 0.513, against 0.641, 0.411 and 0.263 under independence). At 100 samples the fitted law gives 0.244, the false acceptance of 3.2 independent checks.

A.3 Judge and check-writer prompts

The zero-shot τ^2 -bench judge receives the following system prompt:

“You audit the work of an AI customer-service agent after a chat has ended. The agent had to follow the domain policy and do what the customer validly asked for, using tools that read and write the database. You see the policy, the customer’s messages (not the agent’s), every database change the agent made, and the relevant database records as they were before the chat. Decide whether the agent’s work is WRONG: the end state violates the policy, misses or gets wrong something the customer validly requested, or contains a change the customer did not request or confirm. If the work is consistent with the policy and the customer’s requests, it is OK.”

The user turn supplies the policy, customer messages, database changes, and relevant start records. It requests `{"verdict": "WRONG" or "OK", "reason": ...}`.

The peer judge uses the same prompt with this passage before the answer request, followed by each peer’s changes, capped at 4,000 characters:

“Other attempts at the same customer request (unverified). The same customer request was also handled in k other independent chats, by other agents. Below are the database changes each of those attempts made. Their correctness is UNKNOWN: any or all of them may be wrong. They are evidence about what the request involves, not ground truth. The customer’s wording may differ between chats.”

The check writer’s system prompt begins:

“You write verification probes for AppWorld, a simulated world of apps ... A probe is a small deterministic Python function that inspects the world’s database AFTER the agent finished a task and decides whether the agent’s work is WRONG. ... It never sees the agent’s reasoning and never has the answer key.”

The precedent judge and the ACE playbook’s judge share the AppWorld judge’s system prompt.

A.4 Admission gate

For a candidate check c for template T at window t , let S contain up to 400 past successes of T , N up to 150 past successes of other templates, and F up to 300 past failures of T , all from windows $\leq t$. Admit c exactly when it fires on no item of S , on at most 2% of N , and catches at least $\max(1, \text{catches of the current check})$ items of F . Checks seeded from correct attempts before T has failed have no catch requirement. Revoke an admitted check after its third false fire.

A.5 Model calls and tokens

The pass/fail library made 261 Claude Opus 5.5 calls across two arms without errors, using 4.74M input and 0.99M output tokens. The precedent judge made 2,683 calls on 2,749 evaluation items, using 8.9M input, 17.7M cache-read, 95.2M cache-write, and 1.5M output tokens. The ACE playbook made 4,518 calls: 2,690 judge, 914 reflect, and 914 curate calls. Its token counts in the same categories were 57.6M, 12.1M, 42.6M, and 4.9M. The graded run made 192 writer calls serving 261 requests across its arms; the anticipation run made 139 calls. Neither had call errors.

A.6 Preregistration and recomputation

In Table 11, P marks a preregistered contrast; X, an exact recomputation from raw results with an independent implementation; S, re-execution of sampled stored verdicts with identical results; and R, re-tabulation from registered result files. Each judge returns one sample per attempt, and the registered rule uses the first verdict when records are duplicated. The information-matched agreement contrast under the first-verdict, last-verdict and all-records-averaged rules is +16.52, +17.99 and +17.25 points of J ; the zero-shot contrast is +39.99, +38.52 and +39.26.

Table 11: **Reported contrasts and how each was verified.** Claude Opus 5.5 is the judge and check writer throughout. Intervals are 95% bootstraps; library differences are points of catch unless stated otherwise.

Result	n	Estimate [CI]	Verification
Agreement (same 4) — peer judge, held-out τ^2 -bench (ΔJ)	600 attempts, 68 failures, 60 tasks	+16.5 [+5.4, +32.1]	P, X
Agreement (50 panels) — zero-shot judge, held-out τ^2 -bench (ΔJ)	same	+40.0 [+26.0, +61.0]	P, X
Peer judge OR agreement — peer judge (ΔJ)	same	+20.74 [+9.49, +36.01]	R
Peer judge — zero-shot judge (ΔJ)	same	+22.2 [+11.2, +38.2]	R
Agreement (4 attempts) — zero-shot judge, τ^2 -bench training record (ΔJ)	1,133 attempts, 104 tasks	+15.6 [+10.7, +20.8]	X
Judge until three earlier attempts, then agreement — zero-shot judge (ΔJ)	same	+18.0 [+12.3, +23.8]	X
One-sided agreement labels — completion flag (precedent judge; ΔJ)	1,036 failure, 627 correct never-seen states	+64.1 [+52.9, +70.3]	X
Library — copy frozen after 200 failures	1,036 never-seen failure states	+24.0 [+16.4, +32.8]	P, X
Library — copies frozen after 800 and 1,800 failures	759 and 568 states	+12.0 [+5.9, +18.0]; +10.6 [+4.3, +19.2]	R
Library — copy frozen after 50 failures	1,036 states	+76.5 [+65.4, +84.8]	R
Grader text — pass/fail outcomes only	1,036 states	+2.4 [+0.8, +4.4]	X
Library — frozen copy, templates already covered	921 states	+17.6 [+11.1, +24.4]	R
Library — frozen copy, first appearances of a failure mode	171 first appearances	+13.5 [+5.4, +22.8]	R
Library — precedent judge ($p \leq 5$), windows 29–38	345 states	+7.3 [+1.5, +13.4]	X
Library from completion flags, never-seen-state catch	1,036 states	5.1% (90.1% from official outcomes)	P, R
Checks from Claude Haiku 4.5, Qwen 27B and GPT-6 Astra failures, Claude Opus 5.5 and Claude Opus 4.8 never-seen states	32 states; 7,185 correct attempts	30 of 32 caught; 0 false fires	S, X
Anticipate — failure-only library, instance first failures	110 first failures	+55.5 [+44.6, +65.6]	P, S, X
Same, instances whose correct attempt was not shown	44 first failures	+36.4 [+21.7, +51.2]	S, X

B Additional Results

B.1 Grader text adds little

Past pass/fail outcomes carry nearly all of the signal: showing the writer the grader’s report on past failures adds +2.4 [+0.8, +4.4] points of never-seen-state catch, to 92.5%.

B.2 Library composition

The pass/fail library reached 48 checks, one per template, in window 8. Its learning arm proposed 48 mints, of which the gate admitted 35, and 84 maturations, of which it admitted 77; it also admitted 8 of 9 re-mints, 10 of 11 retries and 3

of 3 repairs. One check was revoked, in window 5, after its third false fire. Of the templates, 42 first fail in windows 1–4 and 6 in windows 5–10; none first fails after window 10.

B.3 Per-actor catch in the graded run

On never-seen failure states, the graded-run library caught 82.6% of Claude Haiku 4.5’s (854), 91.1% of Qwen 27B’s (247), and 29 of 29 of Claude Opus 5.5’s. Per attempt, it caught 100% of Claude Opus 5.5’s 53 failures, with no flags on 6,865 correct attempts; it caught 94.2% of Claude Haiku 4.5’s, 95.3% of Qwen 27B’s, and 85.7% of GPT-6 Astra’s 14.

B.4 Actor composition by window bin

Most failures in windows 2–38 come from Claude Haiku 4.5 (2,996 of the 3,538 failures by the models named in Section 4) or Qwen 27B (469). By bin, total failures and Claude Haiku 4.5/Qwen 27B shares are: windows 1–4, 200 (83%/0%); 5–10, 600 (92%/2%); 11–18, 800 (99%/1%); 19–26, 800 (93%/5%); 27–32, 600 (96%/3%); and 33–38, 607 (32%/65%).

B.5 Example check

This check verifies the Spotify template that asks the agent to follow every artist of the songs of one genre in the user’s playlists (Section 6.1). It is the window-8 maturation. The window-4 version differs in a narrower instruction parser and in lacking the condition on the submitted answer. The code is verbatim, except that the regular expression that matches the AppWorld instruction is elided as . . . , blank lines are removed, and wrapped lines and adjacent string literals are joined.

```
import re
def probe(ctx):
    m = re.search(r"...", ctx.instruction, re.I)
    if not m:
        return {"applicable": False, "flag": False, "evidence": "instruction does not match this template"}
    genre = m.group(1).strip().lower()
    S = ctx.models.start.spotify
    E = ctx.models.end.spotify
    me = S.User.find_one(email=ctx.supervisor["email"])
    if me is None:
        return {"applicable": False, "flag": False, "evidence": "supervisor spotify user not found"}
    expected = set()
    for pl in S.Playlist.find_all(user_id=me.id):
        for s in pl.songs:
            if (s.genre or "").strip().lower() == genre:
                expected.update(s.artist_ids or [])
    if not expected:
        return {"applicable": False, "flag": False, "evidence": f"no {genre} songs found in the user's playlists in the start state"}
    start_follow = {f.artist_id for f in S.UserArtistFollowing.find_all(user_id=me.id)}
    end_follow = {f.artist_id for f in E.UserArtistFollowing.find_all(user_id=me.id)}
    missing = sorted(expected - end_follow)
    if missing:
        return {"applicable": True, "flag": True, "evidence": f"artists {missing} of {genre} songs in the user's playlists are not followed; expected all of {sorted(expected)} (check all playlist pages)"}
    removed = sorted(start_follow - end_follow)
    if removed:
        return {"applicable": True, "flag": True, "evidence": f"previously followed artists {removed} were unfollowed"}
    extra = sorted((end_follow - start_follow) - expected)
    if extra:
        return {"applicable": True, "flag": True, "evidence": f"newly followed artists {extra} are not artists of any {genre} song in the user's playlists (expected set {sorted(expected)})"}
    # This is an action task; no answer should be submitted.
    ans = ctx.predicted_answer
    if ans is not None:
        txt = str(ans).strip()
        if txt and txt.lower() not in ("none", "null"):
            return {"applicable": True, "flag": True, "evidence": f"the follows are correct, but this task requires no answer and the agent submitted answer {ans!r}; complete the task without an answer (answer=None)"}
    return {"applicable": True, "flag": False, "evidence": f"all {len(expected)} expected artists {sorted(expected)} are followed, no extra follows, no answer submitted"}
```