

Imitation Holds a Key to Evolution: Localized Repair and Verified Skill Acquisition for Adaptive Agents

Kai Hu¹ Hua Huang²

Abstract

An agent can retain successful trajectories without learning which part of an experience should repair a new discrepancy. We formulate passive imitation (PI) as an architectural loop that localizes a violated contract, extracts a local goal, proposes an effect-matched skill, tests a small patch, and retains evidence after full-task verification. We instantiate this loop as a deterministic symbolic controller and analyze its finite repair-query complexity under explicit identifiability and noninterference assumptions. Across ten simulated tool worlds, PI solves all 600 cold held-out composition tasks with three allowed small trials, compared with 6.17% for a whole-trajectory control, while an equally expressive compositional repair control matches PI. On an unseen tool revision, PI achieves 70.0% success versus 24.0% without small trials and 79.0% with self-trial enumeration; an exploratory 24-task adaptation stream subsequently raises first-submission success to 100%, supporting a limited correction-and-retention mechanism rather than a unique incentive advantage or general intelligence. The CPU-only implementation, integrity checks, and reproducible results are available at <https://github.com/FriendOfYishan/PI>.

1. Introduction

An adaptive agent must decide what a remembered experience licenses it to change. A reporting workflow may select paid records, resolve duplicate revisions, and normalize monetary values. Copying it into a different request can repair one operation while damaging two others. A failed workflow may also contain useful components that support

a smaller intervention.

We ask: *can an agent convert a discrepancy into a locally verified skill that improves later decisions without updating a neural model?* This requires identifying the implicated computation, expressing its desired effect, testing a candidate transformation, and assessing whether the evidence supports reuse. Current-task success and reusable skill acquisition are separate events.

We call this organization *passive imitation*. “Passive” refers to learning opportunities arising during an assigned task. Acquired skills should be used before a discrepancy recurs. Candidates can come from another execution, the agent’s own history, or action enumeration. Imitation is an optional source of hypotheses; the reusable object is a transformation with a proposed effect and an applicability condition.

Existing agents combine reasoning, feedback, memory, and routines: ReAct interleaves reasoning and actions (Yao et al., 2023); Reflexion retains verbal feedback (Shinn et al., 2023); ExpeL extracts experiential insights (Zhao et al., 2024); and Agent Workflow Memory induces workflows (Wang et al., 2025). Neither experience reuse nor localized correction is new in isolation. We make their proposed organization explicit and test its components in an inspectable setting.

Our small symbolic world removes language-model variance while exposing supplied assumptions: typed roles, an executable grammar, and structured assertions. Effect recognition and goal projection are programmed. We test the architecture with these interfaces available, without testing their discovery from unrestricted language.

Local patches outperform whole-trajectory reuse on withheld compositions, and verified records improve future first attempts. Small trials help when tool meanings change but are unnecessary for reliable familiar fragments. A strong compositional control ties PI, while demonstration-free enumeration performs better on unseen semantics under a tight budget. These findings constrain the architecture’s interpretation.

Our contributions are a controller separating retrieval, tentative repair, full verification, and reuse; a conditional query analysis explaining when promotion thresholds are redun-

Author email: Hua Huang <S251101018@std.cqupt.edu.cn>. ¹Beijing International Studies University ²Chongqing University of Posts and Telecommunications. Correspondence to: Kai Hu <2025110321@stu.bisu.edu.cn>.

dant; and an exhaustively checked CPU study with matched budgets, an expressive control, ablations, transfer failures, and exploratory adaptation. Evolution here means cumulative change in a reusable repertoire, without a claim of biological evolution or artificial general intelligence.

2. Related Work

Feedback and agent memory. Reflexion uses linguistic feedback as an experience signal for subsequent attempts, while Self-Refine iterates generation, feedback, and revision without additional model training (Shinn et al., 2023; Madaan et al., 2023). Self-Debugging uses execution and explanations to improve generated programs (Chen et al., 2024). These systems establish that nonparametric correction can be useful. Our controller separates the implicated scope and the desired effect before proposing a patch, and distinguishes local acceptance from evidence for future reuse. We study that separation in a finite interpreter, not as a replacement implementation of any of these LLM systems.

ExpeL stores extracted insights and past experiences for later decisions (Zhao et al., 2024). Agent Workflow Memory induces and retrieves routines from previous executions (Wang et al., 2025). Voyager combines executable skills, environment feedback, and iterative program improvement (Wang et al., 2024). EvolveR further couples distilled principles and empirical utility with policy evolution (Wu et al., 2026). Consequently, neither reusable experience nor an incentive-like utility score is itself a novelty claim. PI is positioned as a transparent composition of mechanisms whose individual roles can be tested, with a particular emphasis on small interventions and the conditions attached to a retained action.

Explanation and repair. Explanation-based generalization connects domain knowledge and an explanation of a specific case to reusable knowledge (Mitchell et al., 1986). Our use of typed lineage and contract projection has a clear conceptual debt to that tradition. It is substantially simpler: the experiment does not infer a domain theory, prove a natural-language explanation, or discover a new predicate vocabulary. Delta debugging isolates failure-inducing input through tests (Zeller & Hildebrandt, 2002). It motivates the importance of narrowing a discrepancy, but our localization is supplied through public data lineage, not computed by a delta-debugging search. Claims about autonomous fault localization would therefore go beyond the experiment.

Imitation and sequential learning. Imitation learning must account for the distribution of states induced by a learner’s actions; DAGger addresses this through dataset aggregation and no-regret learning (Ross et al., 2011). PI does not fit a policy to expert action labels and does not

inherit that guarantee. Its evidence is a table of conditioned actions and trial outcomes, with no gradient updates. Temporally extended skills are also central to the options framework (Sutton et al., 1999); our atomic stage transformations are more restricted than general options. These connections encourage precise terminology: this work concerns architectural experience reuse, rather than a new universal imitation-learning objective.

Scope of the empirical comparison. All evaluated methods share one interpreter. Whole-trajectory replay is restricted to complete workflows and does not reproduce the broader capabilities of the cited LLM agents. The compositional repair control is the informative comparison for benefits attributable uniquely to PI; its tie is reported explicitly.

3. Problem Formulation

3.1. Tasks, Programs, and Feedback

A task is $x = (v, r, \rho, D)$, where v identifies a tool revision, r is a structured request, ρ binds physical fields to logical roles, and D is an input table. A program is a tuple $p = (a_1, \dots, a_m)$ with $a_j \in \mathcal{A}_j$. The environment executes

$$z_0 = \rho(D), \quad z_j = T_{v,j,a_j}(z_{j-1}), \quad j = 1, \dots, m. \quad (1)$$

The action identifiers are public, but their revision-dependent implementations are not available to the controller as a lookup table. Execution returns an artifact, a before/after trace, and feedback. The public request specifies a local goal $g_j = r_j$. A stage contract $\phi_j(z_{j-1}, z_j; g_j)$ states whether that transformation has the requested effect on its actual input.

Let $b_j \in \{0, 1\}$ denote the corresponding contract result and let $F(b) = \sum_j (1 - b_j)$ count failed contracts. A full audit additionally compares the final aggregate and supporting row identities with an independent reference computation. Its acceptance bit $A(x, p)$ requires both the artifact checks and every stage contract to pass. The controller receives no correct action identifier, hidden registry, or reference answer from the feedback channel. Nevertheless, the feedback is strong: structured diagnostics name a violated output path, and a public lineage map identifies its producer.

This distinction matters for interpreting observability. The controller does not observe latent tool semantics directly, but the benchmark is not a general partially observable decision process with a learned belief state. Its hidden uncertainty is a finite action-to-effect permutation. Exposing logical field roles and contracts removes important sources of difficulty that would occur in a natural document workflow.

3.2. Experience and the Unit of Reuse

A source demonstration contains a task, an accepted program, and its stagewise trace. From a transition (z_{j-1}, a_j, z_j) , an effect recognizer infers $e_j = \psi_j(z_{j-1}, z_j)$. A reusable fragment is

$$f = (v, j, e_j, a_j, \text{source identifier}). \quad (2)$$

The recognizer uses observable set changes, timestamps, and numerical ratios. It never consults the environment’s hidden permutation. In this implementation its effect vocabulary is predefined; abstraction means mapping concrete changes into that vocabulary, not inventing a new concept.

Persistent acquired evidence is indexed by a condition $c = (v, j, g_j)$ and an action a . A record $E(c, a) = (s, f, R, U)$ stores successful and failed trial evidence, cumulative incentive, and a set of distinct fully accepted task identifiers. This representation preserves an action’s intended role and revision. It does not preserve enough information to certify behavior under arbitrary distribution shift. Source fragments and acquired evidence also have different status: an old fragment may propose a candidate in a new revision, whereas proactive reuse requires confirmations under the current revision key.

3.3. Budget and Evaluation Objectives

Each episode permits at most K small trials and two full submissions. The first full submission exposes any discrepancy. After bounded correction, a second submission checks the complete task. A small trial executes a proposed program on a shared subset $\mu(D)$ comprising two complete order histories; the controller changes one stage in that program, rather than executing only the changed stage.

We report final acceptance, first-submission acceptance, the number of small trials, and row-stage visits. The latter counts rows processed by stage operators over full and small executions, plus the rows consumed by the final aggregate/emission step. It excludes retrieval, trace parsing, controller computation, and corpus construction. It is consequently an interpreter-work proxy, not a measurement of tokens, latency, energy, or monetary cost. No fixed long-term memory or context-window constraint is imposed in the present study. Keeping these quantities separate prevents an execution-saving result from being misrepresented as a language-model efficiency result.

4. Passive Imitation Architecture

4.1. From a Discrepancy to a Local Candidate

Figure 1 illustrates the controller. For a failed diagnostic d , localization maps its public output path to a producing stage $j = G(d.\text{path})$. Goal projection then selects $g_j =$

r_j . The textual explanation is retained for inspection, but its semantics are not parsed by a language model. The diagnostic path and structured request perform the operative reasoning in this pilot.

Candidate construction first considers actions with positive acquired evidence for (v, j, g_j) . It then selects source fragments with the same stage and inferred effect as the local goal, preferring the current revision, followed by a heuristic evidence score and a deterministic source-identifier tie break. All remaining typed actions are appended, with duplicates removed. This final enumeration step makes source demonstrations optional. The controller skips the current program and previously tried pairs of stage and full program signature within the episode.

The candidate transformation is transplanted into the current program only at stage j . It is not a complete trajectory replacement. That distinction is the main structural contrast with the whole-trajectory and no-localization controls. A fragment proposes an action that may meet a desired postcondition; it does not provide authority to overwrite unrelated operations or to bypass verification.

4.2. Small Trials and Full Verification

For proposal $p' = p[j \leftarrow a]$, the environment returns microtask feedback b' . The patch is tentatively committed when the target contract passes. If it fails, the current program is retained and the next distinct candidate is tried. The controller always chooses the first currently failed diagnostic in the fixed stage order. It stops when all current contracts pass, the trial budget is exhausted, or no eligible proposal remains. A final full-task audit is performed after any initially failed submission, even if microtask feedback appears satisfactory.

This design separates hypothesis testing from promotion. A small test can reject a candidate cheaply, and in the constructed vocabulary it identifies each valid transformation. Outside that vocabulary it is only evidence on a subset. The full audit therefore remains part of the controller. Its role cannot be replaced by a positive incentive or a plausible explanation. In the implementation, tentative commitment depends on the target contract, not on the sign of a reward.

4.3. Incentive, Retention, and Proactive Reuse

For a microtask with n_μ input rows and execution work W_t , the recorded incentive is

$$r_t = F(b) - F(b') - 0.01 \frac{W_t}{\max(1, n_\mu)}. \quad (3)$$

If the target contract fails, the stored value is $\min(0, r_t)$. Successful and failed target trials increment their corresponding evidence counts during acquisition. A fully accepted task supplies one additional success and one unit of

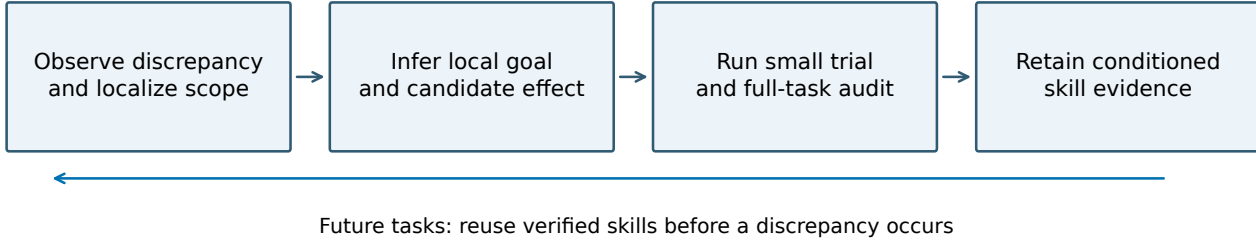


Figure 1. The implemented passive-imitation loop. Structured feedback and public lineage identify the local scope; symbolic effect recognition supplies candidate skills. A local trial pass permits a tentative patch. Full-task acceptance contributes evidence for later proactive reuse. Candidate sources may include other executions or the controller’s own trials.

incentive to each action in its final program; its identifier is added to U without duplication. Full-task rejection withholds those confirmations. It does not itself add a failure count to every stage.

The reuse score is

$$q(c, a) = \frac{s(c, a) + 1}{s(c, a) + f(c, a) + 2}. \quad (4)$$

An action may enter the initial program of a later task when $|U(c, a)| \geq 2$ and $q(c, a) \geq 0.7$. Eligible actions are ordered by q , with cumulative incentive as a tie break. The score has a smoothed-count form, but we do not interpret it as a calibrated posterior probability: microtask outcomes and full-task confirmations can be correlated and can arise within the same episode. The two-task requirement counts distinct task identifiers, not statistically independent proofs of reliability.

At frozen evaluation time, within-episode repair is permitted, but E is not updated. Acquisition and evaluation are separate calls. This boundary is enforced by state hashes before and after each frozen batch. Resetting the retained evidence restores the cold controller. The architecture can therefore improve future first submissions through explicit experience accumulation, rather than through hidden changes during testing.

5. Conditional Analysis

5.1. When Local Repair Has a Finite Query Bound

The following analysis describes a finite interface, not unrestricted software repair. Let J be the stages failing in the initial program. Three assumptions specify when local evidence is sufficient.

Assumption 1 (Faithful Localization). *Every violated stage contract identifies its producing stage through G , and every incorrect stage is exposed by a violated contract. The*

diagnostic order visits an exposed incorrect stage until it is repaired.

Assumption 2 (Stable Identifiability). *For each condition (v, j, g) there is a unique correct action. For all inputs reachable during repair, that action passes the stage contract and every other action fails it. The shared microtask and full task agree on these contract outcomes.*

Assumption 3 (Complete Local Search). *Candidates cover every alternative action in $\mathcal{A}_j$. Trials alter only the selected action, and failed candidates are not retried at that stage before a successful repair.*

Proposition 1 (Finite Local Repair). *Under Assumptions 1–3, a correct program is found using at most*

$$Q \leq \sum_{j \in J} (|\mathcal{A}_j| - 1) \quad (5)$$

small trials and two full submissions, provided $K \geq Q$.

Proof. At an incorrect stage, the current action is already known to fail. Exhausting its alternatives requires at most $|\mathcal{A}_j| - 1$ queries and includes the unique correct action. Identifiability ensures that only this action is committed. Locality leaves other action choices unchanged, and stability ensures that a later change in their inputs cannot invalidate a correct action. Thus a repaired stage does not become incorrect again. Summing over initially incorrect stages proves the bound. The initial and final full submissions expose the initial discrepancy and verify the repaired complete program. $\square$

For the present action-set sizes $(2, 3, 3)$, the worst-case bound is five small trials. The empirical budget of six therefore exceeds this bound. A budget of three need not suffice when candidate ordering is uninformed. On familiar revisions, effect-matched candidates often place the correct action first, reducing the number of needed queries to the

Algorithm 1 Passive Imitation Episode

Require: Task x , persistent evidence E , fragments $\mathcal{F}$, budget K , acquisition flag L

```

1:  $p \leftarrow \text{Initialize}(x, E)$ ;  $y \leftarrow \text{FullAudit}(x, p)$ 
2:  $b \leftarrow y.\text{contracts}$ ;  $\mathcal{H} \leftarrow \emptyset$ ;  $\mathcal{U} \leftarrow []$ 
3: if  $y.\text{accepted}$  then
4:   if  $L$  then
5:     Record full-task confirmation for  $p$  in  $E$ 
6:   end if
7:   return  $(p, y, E)$ 
8: end if
9: for  $t = 1$  to  $K$  do
10:  if all contracts in  $b$  pass then
11:    break
12:  end if
13:   $d \leftarrow$  first violated diagnostic;  $j \leftarrow G(d.\text{path})$ ;  $g \leftarrow r_j$ 
14:   $a \leftarrow$  next candidate from  $E, \mathcal{F}, \mathcal{A}_j$ , excluding repeats and current program
15:  if no candidate exists then
16:    break
17:  end if
18:   $p' \leftarrow p[j \leftarrow a]$ ; add  $(j, p')$  to  $\mathcal{H}$ 
19:   $y' \leftarrow \text{SmallTrial}(\mu(x), p')$ ; compute  $r_t$  by Eq. 3
20:  Append  $(j, a, \text{target pass}, r_t)$  to  $\mathcal{U}$ 
21:  if the target contract passes in  $y'$  then
22:     $p \leftarrow p'$ ;  $b \leftarrow y'.\text{contracts}$ 
23:  end if
24: end for
25:  $y \leftarrow \text{FullAudit}(x, p)$ 
26: if  $L$  then
27:  Update  $E$  from  $\mathcal{U}$ ; if  $y$  is accepted, add distinct full-task confirmations for  $p$ 
28: end if
29: return  $(p, y, E)$ 
```

number of incorrect stages. The cold composition result is consistent with this mechanism, rather than evidence of solving a large unconstrained search problem.

The strong assumptions explain the limits of the guarantee. A contract on a final aggregate alone may not expose which stage is wrong. A tiny fixture may fail to distinguish actions on rare cases. An upstream patch may change the meaning of a downstream predicate. In any of these situations the proof no longer applies. Our exhaustive checks verify micro/full agreement for all 18 programs on every generated primary task; they do not establish stable identifiability for programs or inputs outside this generated set.

5.2. Why an Incentive Threshold Can Be Redundant

Proposition 2 (Redundant Promotion Threshold). *Suppose accepted tasks certify the unique correct action for a con-*

dition, and that action never receives failed local evidence. After two distinct full-task confirmations its score in Eq. 4 is at least $3/4$. A threshold of 0.7 therefore imposes no additional promotion delay relative to the two-task condition.

Proof. Two confirmations imply $s \geq 2$ and the premise gives $f = 0$. Hence $(s + 1)/(s + 2) \geq 3/4 > 0.7$. Uniqueness prevents a second incorrect action from receiving valid full-task confirmations for the same condition. $\square$

This proposition is about eligibility, not a universal equivalence between two controllers. Candidate ordering before promotion can still differ, and noisy tests can invalidate the premise. It nevertheless explains why the numerical threshold has little opportunity to help in a deterministic world with perfect diagnostic separation. The strong control removes that threshold and removes evidence-score ranking within source fragments, but shares the remaining controller machinery. The reported tie is therefore plausible and substantively important.

5.3. What Verification Does Not Guarantee

Two accepted tasks do not certify distribution-free reliability. An action can pass sampled inputs and fail elsewhere. Adaptive selection and correlated tests also preclude simply multiplying per-test error probabilities. The evidence score therefore supplies no calibrated probability of future correctness.

Version keys prevent automatic promotion across versions but fragment experience when versions share semantics. Cross-version fragments remain hypotheses requiring tests. This pilot measures consequences of that reuse-caution tradeoff without optimizing applicability conditions.

6. Experimental Design

6.1. A Finite Reporting Environment

The environment has three ordered stages. Population selection either keeps all records or only settled records. Revision resolution keeps all revisions or chooses the oldest or latest revision per order. Amount normalization interprets source values as yuan, cents, or hundreds of yuan. Actions have opaque identifiers $\{F0, F1\}$, $\{D0, D1, D2\}$, and $\{N0, N1, N2\}$. Each simulated world independently permutes the meaning of these identifiers for revisions A, B, and C.

Ordinary tasks contain six orders with two revisions each. Larger schema-shift tasks contain eight orders with three revisions each and allow settlement status to change across revisions. Column names vary, but a public role mapping is always supplied. Schema shift thus tests execution under changed layouts and row structure with known roles; it

does not test autonomous schema understanding. Every task requests an exact final total and the identities of its supporting rows.

The source corpus contains 24 accepted demonstrations per world: two familiar revisions, six atomic/default requests, and two separately generated examples for each pair. A default request asks for all records, all revisions, and yuan. Each of the other five atomic requests changes one of these goals. Source transitions are converted into 16 distinct revision-conditioned fragments. Complete requests with at least two nondefault goals are withheld from both demonstrations and the initial acquisition stream. There are 12 such combinations.

The microtask contains the first two order identifiers and all their revisions: four rows ordinarily and six rows in the larger tasks. This is a deliberately identifying fixture for the finite operator vocabulary. It is shared by all controllers, is not selected using a hidden correct-action label, and is checked exhaustively against full-task contract results. The ease of constructing it is part of the benchmark design, not an autonomous experimental-design achievement.

6.2. Primary Protocol and Controls

We fix ten seeds: 11, 29, 47, 61, 83, 101, 127, 149, 173, and 197. Each seed defines one tool world. Acquisition supplies 96 atomic tasks, consisting of eight instances of each familiar revision/request pair. Final evaluation has 240 tasks per world, evenly divided among new atomic instances, held-out compositions, larger schema-shift compositions, and unseen-revision-C compositions. All controllers see the same tasks and public information. The local protocol was recorded before the primary run; it was not publicly preregistered.

Cold controllers are evaluated at $K = 3$. After acquisition, frozen controllers are evaluated at $K \in \{1, 3, 6\}$. Separate 24-task frozen probes are reused at acquisition checkpoints 0, 24, 48, and 96. The final task identifiers are disjoint from acquisition, source, and probe identifiers. Reusing probes across checkpoints produces repeated measurements, not independent samples. A reset condition discards acquired evidence and reevaluates PI.

Eight controllers isolate the relevant operations. *Whole-trajectory replay* proposes only complete source programs, retains accepted complete workflows by exact request and revision, and tests proposals with the same microtask budget. *Full PI* implements Algorithm 1. *No localization* uses the diagnosed goal to rank complete templates but replaces the whole program. *No goal matching* preserves local scope while ranking raw templates by request similarity instead of matching inferred transition effects. *No small trial* directly adopts the highest-ranked local candi-

dates and proceeds to the second full audit. *No retention*, named `pino.incentive` in code, disables all persistent learned-action evidence. *Compositional repair* preserves local goal-directed testing and two-task confirmation while removing the additional score threshold and source-fragment evidence ranking. *Self-trial only* enumerates typed actions without consulting demonstrations, while preserving localization, verification, and learning.

The no-retention condition is not a clean ablation of a reward formula: it removes cross-task verified reuse altogether. The strong compositional control is needed to assess whether the extra numerical machinery contributes beyond local repair and confirmation. It still shares confidence-based tie ordering in some helper functions; it is a targeted control, not an independently engineered competing system. These details are exposed in the implementation appendix.

6.3. Measurement and Integrity

Final success is the fraction of tasks accepted by the independent full evaluator. First success measures acceptance before current-episode correction. Final collateral regression occurs when a contract passing initially fails at the final audit. This differs from the number of regressions across tentative commits, which is logged separately. Mean row-stage visits includes unsuccessful attempts and trial work; lower cost is not automatically preferable if success also falls.

We aggregate 60 final tasks within each seed/group, then summarize the ten seed values. Since group sizes are equal, the mean equals the pooled rate across 600 tasks. Where reported, standard deviations describe variability across the ten worlds, not an independent-task confidence interval. Common tasks make method comparisons paired; task counts must not be treated as 600 independent tool environments. We use descriptive comparisons without selecting a statistical significance threshold.

Before score interpretation, the integrity suite checks 3,600 generated tasks and all 18 programs per task, for 64,800 full/micro comparisons. It verifies a unique accepting program, the rejection of corrupted totals and provenance, source-effect inference, disjoint identifiers, intervention scope, budget limits, and frozen-state hashes. After the first run, a logging-only omission in the trajectory control’s tentative-regression counter was corrected. The experiment was rerun; every other summary field was verified unchanged. The public release was rerun again after changing only output paths, and all eight deterministic summary/check files were byte-identical to the corrected original outputs.

7. Results

7.1. Local Repair Enables Withheld Compositions

Table 1 reports cold composition results at $K = 3$. PI reaches 100% final success, using 1.74 small trials on average. Whole-trajectory replay remains at 6.17%, the same rate as its initial default program. This large gap has a specific interpretation: none of the observed atomic complete programs is an exact solution to a genuinely multi-goal familiar-revision request. The replay controller cannot assemble missing combinations. Its residual successes arise from the default opaque identifiers happening to implement the requested combination in a particular world.

Removing localization reduces success to 9.50% and produces 41.83% final collateral regression. A complete replacement can satisfy the currently targeted goal while breaking a previously correct stage. Locality prevents that behavior in this finite world. Removing effect-to-goal matching has a smaller effect: success falls to 92.50%, with 1.90 trials. Raw request similarity still contains useful information, so this condition does not remove all goal information or all reasoning about the request.

The no-small-trial condition also solves all cold familiar compositions. Same-revision effect-matched fragments already propose correct local actions, making validation redundant for these tasks. Its work is 58.82 row-stage visits versus 75.90 for PI. A claim that small trials always improve efficiency would therefore contradict the measured behavior. Their benefit must be assessed in environments where proposal reliability is lower.

The strong compositional repair control matches PI in success, trials, work, and collateral regression in the reported settings. This comparison prevents attributing the composition result uniquely to the incentive score. Self-trial enumeration, which uses no demonstrations, achieves 80.33% cold success and also has zero collateral regression. Demonstrations improve candidate ordering on familiar semantics, but are not a necessary prerequisite for correction or subsequent learning.

7.2. Retention Changes Future First Submissions

After 96 atomic acquisition tasks, PI solves every held-out composition on the first submission. The same is true of the strong control and self-trial controller. They have acquired componentwise associations that can be recombined into complete requests never observed in acquisition. The no-localization and no-goal variants also reach 100% first success after this repeated atomic stream; their cold repair weaknesses do not prevent eventual acquisition in the simpler training regime.

Disabling retention preserves final success at 100% on fa-

miliar compositions but leaves first success at 6.17%. It repeatedly performs the same 1.74-trial correction. Mean evaluation work is 75.90 row-stage visits without retention and 28.98 with acquired PI skills, a reduction of 61.82%. This is an evaluation-phase work reduction. Acquisition, demonstrations, and retrieval costs are not amortized into that percentage. It does not establish an end-to-end token saving or a favorable break-even point for an LLM implementation.

Whole-trajectory replay remains at 6.17% on withheld compositions, as expected from its restricted representation. Local verified associations instead transfer across recombinations while preserving the frozen-state boundary; this comparison does not establish a failure of modern memory-based agents.

7.3. New Tool Semantics Expose Negative Transfer

Revision C permutes action meanings independently of A and B. Acquired records use version-conditioned keys, so they do not authorize proactive reuse in C. Retrieved source fragments nevertheless influence the order in which opaque actions are tested. At $K = 3$, PI obtains 70.0% final success compared with 24.0% for direct adoption without small trials. Validation and local search repair many proposals whose old meanings are no longer reliable.

However, self-trial enumeration reaches 79.0%, and removing goal-based fragment matching reaches 76.0%. Thus the source-induced ordering is not uniformly useful. An action identifier associated with the desired effect in another revision can consume a scarce trial without carrying predictive information about C. Across-world variation is substantial: the standard deviation is 21.20 percentage points for PI, versus 9.91 for self-trial enumeration. A pooled mean alone hides this variation in transfer difficulty.

Figure 2, left, shows the budget dependence. At $K = 1$, PI reaches 21.5%, below the 24.0% no-trial condition. At $K = 6$, PI, compositional repair, and self-trial enumeration all reach 100%, consistent with the finite-search bound. Self-trial still uses fewer trials on average in this setting: 2.46 versus 2.67 for PI. These results identify candidate ordering as a meaningful failure point rather than supporting an unconditional advantage for imitation.

7.4. Exploratory Adaptation to the New Revision

The primary frozen experiment cannot measure learning across C tasks because its persistent state is intentionally fixed. After examining those results, we therefore declared a separate exploratory follow-up. It preserves the original seeds, tools, final C tasks, and $K = 3$. Each controller first rebuilds its A/B history, then acquires experience on 24 separately generated C tasks: two instances of each multi-

Table 1. Primary results at $K = 3$. Cold composition columns use no acquired action evidence. Learned first success follows 96 atomic acquisition tasks. Unseen C is evaluated with frozen A/B-acquired evidence. Percentages are means over ten worlds, 60 tasks per group per world. Parentheses in the last column give the across-world standard deviation in percentage points. “No retention” corresponds to `pi_no_incentive`.

Controller	Cold held-out composition			Learned composition	Unseen C
	Final success	Small trials	Collateral	First success	Final success (SD)
Whole-trajectory replay	6.17	2.82	12.83	6.17	22.67 (12.02)
Full PI	100.00	1.74	0.00	100.00	70.00 (21.20)
No localization	9.50	2.82	41.83	100.00	19.17 (10.10)
No goal matching	92.50	1.90	0.00	100.00	76.00 (19.57)
No small trial	100.00	0.00	0.00	100.00	24.00 (14.93)
No retention	100.00	1.74	0.00	6.17	70.00 (21.20)
Compositional repair	100.00	1.74	0.00	100.00	70.00 (21.20)
Self-trial only	80.33	2.11	0.00	100.00	79.00 (9.91)

goal request, shuffled independently. Their task identifiers and rows are separate from the final evaluation tasks. Four controllers are evaluated at 0, 6, 12, and 24 C exposures.

For PI, first-submission success rises from 6.17% to 54.33%, 92.33%, and 100%. Final success rises from 70.0% to 96.5%, 100%, and 100%. Mean work falls from 82.19 to 49.40, 31.47, and 28.93 row-stage visits. The strong compositional control follows the same curve. Self-trial reaches 37.33%, 91.17%, and 100% first success at the three nonzero checkpoints. No retention remains at 6.17% first success and 70.0% final success throughout.

These results demonstrate accumulation of verified mappings in a newly encountered but subsequently learned revision. They do not show zero-shot transfer after 24 exposures, nor do they isolate a benefit from numerical incentives over simpler verified reuse. Because this follow-up was designed after the primary analysis, it is reported as exploratory instead of being presented as a prespecified confirmatory experiment.

7.5. An Inspectable Correction Trace

One archived composition task in seed 11 begins with ($F0$, $D0$, $N0$) and violates all three contracts. The feedback identifies the selected-record path; goal projection requests all records. A fragment proposes $F1$, and the microtask confirms that local effect. The remaining discrepancies then lead to $D1$ for oldest revisions and $N2$ for values denominated in hundreds of yuan. The recorded incentives are 0.96, 0.97, and 0.97 as the failed-contract count falls from three to zero. A full-task audit accepts the final program.

The trace exposes each supplied mechanism: path-based localization, symbolic goal projection, programmed effect recognition, trial evidence, and full confirmation. Acquisition updates future preferences; frozen testing does not. The

complete example and implementation are released.

8. Discussion and Limitations

8.1. What the Study Establishes

The study supports local repair and conditioned retention when interfaces expose the relevant factors. The query analysis explains their efficiency and the strong control’s identical results. It does not establish an advantage for a unique incentive mechanism.

Perfectly identifying tests make promotion scores largely redundant. The no-retention ablation tests retained knowledge rather than reward shape, while negative transfer shows that an effect label does not justify reusing an opaque action across versions.

Self-improvement here means that earlier accepted tasks modify persistent state and improve later first submissions. It does not mean discovering new algorithms, expanding neural reasoning capacity, or adapting to arbitrary goal and tool changes.

8.2. Threats to External Validity

First, this is one synthetic domain with 18 possible complete programs and predefined stage boundaries. The environment was designed to make composition informative. A whole-trajectory-only controller consequently has a representational disadvantage. The results cannot be generalized to all retrieval methods, much less to existing LLM agents with unrestricted planning and code generation.

Second, localization and abstraction receive unusually informative interfaces. Public role bindings, typed lineage, and stage assertions make scope selection inexpensive and largely deterministic. Natural tasks may expose only a vague complaint or a final incorrect answer. The ability to infer

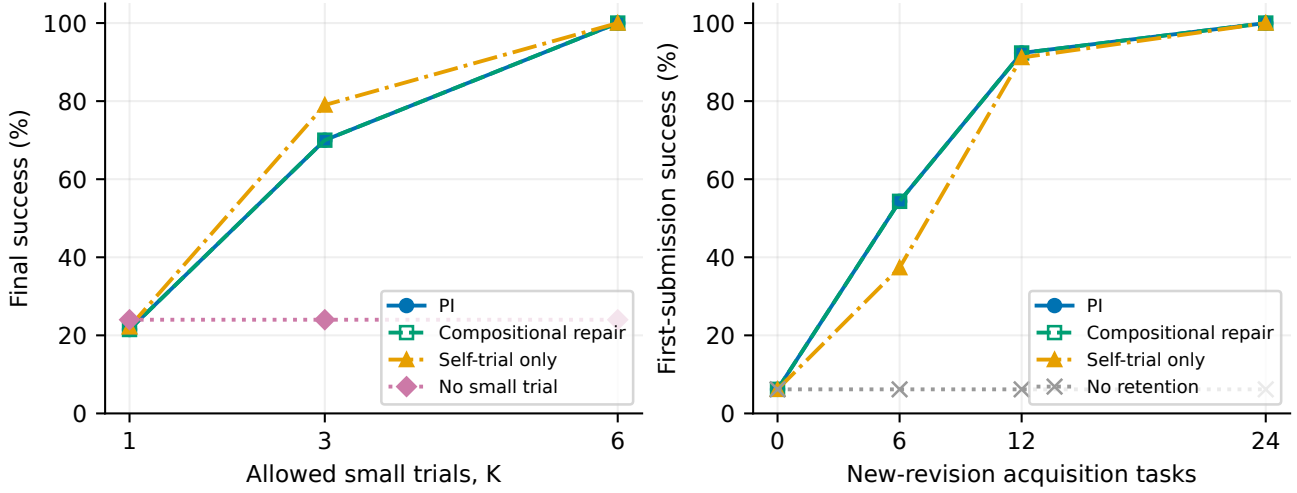


Figure 2. Left: final success on previously unseen revision C after A/B acquisition, with persistent state frozen. The no-trial controller does not use K . Right: first-submission success after separate C acquisition tasks in the exploratory follow-up. Lines show means over ten worlds; PI and compositional repair overlap. Repeated evaluations use the same frozen final C tasks. Learning C changes the interpretation from unseen-revision transfer to retention within a learned revision.

a correct local goal from that feedback is not tested here. In particular, no claim about detecting or resolving natural-language hallucinations follows from these experiments.

Third, microtests are diagnostic by construction. Their agreement with the full task is exhaustively verified in the finite generated world, but their reliability would be unknown for an arbitrary program, external API, or changing dataset. The full evaluator also has access to a reference computation that may be unavailable in practice. Plausible-looking feedback and a positive incentive cannot substitute for that missing verifier.

Fourth, the study does not constrain persistent storage, bound prompt length, or measure language-model tokens. The evidence table retains task identifiers, and its size can grow with the acquisition stream. A bounded-memory implementation would need to decide which conditions, failures, and confirmations can be merged or discarded. Our measurements establish only that learned reuse can reduce repeated interpreter work in evaluation.

Finally, the thresholds, cost coefficient, candidate-ordering rules, and fixture design are not validated across a broad family of environments. We report fixed seeds and preserve the original protocol, but do not claim public preregistration or a comprehensive hyperparameter study. The online follow-up is exploratory, and the ten tool worlds share the same hand-designed task generator. Both facts limit the strength of extrapolation.

8.3. Implications for Architecture Research

A stronger test would require scope, goal, or applicability inference under imperfect feedback, with independently measured outcomes. It should retain an equally expressive repair-and-reuse control and a demonstration-free condition, since modularity and enumeration already explain substantial present benefits.

An LLM implementation should charge diagnosis, candidate generation, trial interpretation, memory maintenance, and failed attempts to a matched total budget. Token-saving claims require measured totals and acquisition-inclusive success–cost comparisons; those results are outside this study.

9. Conclusion

Passive imitation turns structured discrepancies into local repair hypotheses and accumulates condition-dependent verified skills. In a finite reporting environment, locality enables withheld compositions and retention improves future first submissions. Trials help unreliable proposals, whereas familiar fragments need no trials and source ordering can hurt under semantic changes. A compositional control matches PI, supporting the shared mechanism without a unique incentive advantage. Released code, exhaustive checks, and negative findings provide a basis for testing learned, uncertain interfaces.

Impact Statement

This work studies a small synthetic controller and makes no deployment claim. Verified local reuse could reduce repeated mistakes, but incorrect applicability conditions can propagate failures across tasks. In consequential settings, small tests must not be interpreted as guarantees, and retained behavior must remain subject to appropriate full-task checks. All experimental records are synthetically generated; no personal documents, user feedback, or human-subject data are used. Public code and explicit reporting of baseline ties and transfer failures are intended to make unsupported extrapolations easier to identify.

References

- Chen, X., Lin, M., Schärli, N., and Zhou, D. Teaching large language models to self-debug. In *International Conference on Learning Representations*, 2024. URL <https://openreview.net/forum?id=KuPxiIqPiq>.
- Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegrefe, S., Alon, U., Dziri, N., Prabhunoye, S., Yang, Y., Gupta, S., Majumder, B. P., Hermann, K., Welleck, S., Yazdanbakhsh, A., and Clark, P. Self-Refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/91edff07232fb1b55a505a9e9f6c0ff3-Abstract-Conference.html.
- Mitchell, T. M., Keller, R. M., and Kedar-Cabelli, S. T. Explanation-based generalization: A unifying view. *Machine Learning*, 1:47–80, 1986. doi: 10.1007/BF00116250. URL <https://link.springer.com/article/10.1007/BF00116250>.
- Ross, S., Gordon, G., and Bagnell, D. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pp. 627–635. PMLR, 2011. URL <https://proceedings.mlr.press/v15/ross11a.html>.
- Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., and Yao, S. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.
- Sutton, R. S., Precup, D., and Singh, S. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1–2): 181–211, 1999. doi: 10.1016/S0004-3702(99)00052-1. URL <https://www.sciencedirect.com/science/article/pii/S0004370299000521>.
- Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., and Anandkumar, A. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*, 2024. URL <https://openreview.net/forum?id=ehfRiF0R3a>.
- Wang, Z. Z., Mao, J., Fried, D., and Neubig, G. Agent workflow memory. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 63897–63911. PMLR, 2025. URL <https://proceedings.mlr.press/v267/wang25bx.html>.
- Wu, R., Wang, X., Mei, J., Cai, P., Fu, D., Yang, C., Wen, L., Yang, X., Shen, Y., Wang, Y., and Shi, B. From interactions to principles: Experience-driven self-distillation for evolving LLM agents. In *Proceedings of the 43rd International Conference on Machine Learning*, volume 306 of *Proceedings of Machine Learning Research*, pp. 137024–137044. PMLR, 2026. URL <https://proceedings.mlr.press/v306/wu26bf.html>.
- Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.
- Zeller, A. and Hildebrandt, R. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002. doi: 10.1109/32.988498. URL <https://doi.org/10.1109/32.988498>.
- Zhao, A., Huang, D., Xu, Q., Lin, M., Liu, Y.-J., and Huang, G. ExpeL: LLM agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19632–19642, 2024. doi: 10.1609/aaai.v38i17.29936. URL <https://ojs.aaai.org/index.php/AAAI/article/view/29936>.

A. Implementation and Reproduction

This appendix contains the complete three-script implementation used for the public portable release. Python 3.10 or newer is sufficient; the executed release used Python 3.12. No neural model, GPU, third-party Python package, or network connection is required for the experiment itself. Run `verify_correction.py`, then `run_correction.py`, then `run_online_adaptation.py`. Outputs are written to a `results` directory beside the scripts. Run the primary script before the follow-up so that this directory exists.

The main run executes 94,595 episodes: 79,200 final/reset evaluations, 7,680 acquisition episodes, 7,680 checkpoint evaluations, and 35 detailed examples. The follow-up executes 14,400 episodes, including reconstruction of A/B histories. These totals include repeated evaluation conditions and are not counts of independent tasks. The portable rerun took approximately 21.93 seconds for the primary experiment and 3.49 seconds for the follow-up on the local CPU; these unprofiled measurements are supplied for practical reproduction only, not as comparative runtime results. Integrity checks run separately.

The output directory adjustment is the only change between the corrected original controller and the public release. Eight deterministic result/check files were compared byte for byte. Execution-time metadata and the script checksum are expected to change across releases or runs. Source hashes and the result comparison are recorded in `REPRODUCIBILITY.json` at the repository. The archived local protocol predates the primary run but was not externally timestamped as a preregistration. Its initial description of compositional repair as a simple lookup should be read with the exact shared helper functions below: the control removes the promotion score threshold and source-fragment score ranking, while retaining shared ordering in other paths.

A.1. Environment, Controllers, and Primary Experiment

```

1  """CPU-only PI correction-loop benchmark. All scores arise from execution.
2
3  Policy inputs contain public requests, anonymous tool action IDs, observed
4  traces and assertion explanations. Hidden registries are used only by tools,
5  demonstrators and the independent evaluator, never read by policy code.
6  """
7  from __future__ import annotations
8  import copy
9  import csv
10 import hashlib
11 import itertools
12 import json
13 import random
14 import statistics
15 import time
16 from collections import defaultdict
17 from dataclasses import asdict, dataclass
18 from pathlib import Path
19
20 ROOT = Path(__file__).resolve().parent
21 OUT = ROOT / 'results'
22 STAGES = ('population', 'revision', 'amount')
23 ACTION_IDS = {'population': ('F0', 'F1'), 'revision': ('D0', 'D1', 'D2'),
24               'amount': ('N0', 'N1', 'N2')}
25 EFFECTS = {'population': ('all', 'settled'), 'revision': ('all', 'oldest', 'latest'),
26            'amount': ('yuan', 'cent', 'hundred_yuan')}
27 DEFAULT_GOALS = {'population': 'all', 'revision': 'all', 'amount': 'yuan'}
28 DEFAULT_PROGRAM = {stage: ACTION_IDS[stage][0] for stage in STAGES}
29 SEEDS = [11, 29, 47, 61, 83, 101, 127, 149, 173, 197]
30 METHODS = ['trajectory_replay', 'pi_full', 'pi_no_localization', 'pi_no_goal',
31            'pi_no_trial', 'pi_no_incentive', 'compositional_repair', 'self_trial_only']
32 ASSERTIONS = {
33     'population_constraint': {'path': 'selected_record_ids',
34                              'explanation': "Included records violate the report's requested population."},
35     'revision_constraint': {'path': 'resolved_record_ids',
36                             'explanation': "Chosen revisions violate the requested identity/revision rule."},
37     'value_constraint': {'path': 'normalized_amounts',
38                          'explanation': "Displayed amounts do not preserve value under the declared source denomination."}}
39 # Public typed pipeline data lineage, not generator-provided fault locations.
40 LINEAGE = {'selected_record_ids': 'population', 'resolved_record_ids': 'revision',
41            'normalized_amounts': 'amount'}
42
43
44 def stable_int(value):
45     return int(hashlib.sha256(value.encode()).hexdigest()[:16], 16)
46

```

```

47
48 @dataclass
49 class Task:
50     task_id: str
51     group: str
52     tool_revision: str
53     request: dict
54     roles: dict
55     rows: list
56
57
58 def make_task(seed, task_id, group, revision, request, shifted=False):
59     r = random.Random(stable_int(f'{seed}|{task_id}|rows'))
60     roles = {name: f'{name}_{r.randrange(10**9):09d}' for name in
61              ('amount', 'order', 'settled', 'updated', 'row_id')}
62     rows = []
63     for order in range(8 if shifted else 6):
64         for version in range(3 if shifted else 2):
65             rows.append({roles['amount']: r.randrange(100, 20000) + version,
66                         roles['order']: f'order{order}',
67                         roles['settled']: (order + version) % 2 == 0 if shifted else order % 2 == 0,
68                         roles['updated']: version,
69                         roles['row_id']: f'{task_id}:order{order}:v{version}'})
70     r.shuffle(rows)
71     return Task(task_id, group, revision, dict(request), roles, rows)
72
73
74 def atomic_requests():
75     requests = [dict(DEFAULT_GOALS)]
76     for stage in STAGES:
77         for goal in EFFECTS[stage]:
78             if goal != DEFAULT_GOALS[stage]:
79                 request = dict(DEFAULT_GOALS); request[stage] = goal; requests.append(request)
80     assert len(requests) == 6
81     return requests
82
83
84 def acquisition_tasks(seed):
85     specs = [(revision, request) for revision in ('A', 'B') for request in atomic_requests()] * 8
86     random.Random(seed + 1000).shuffle(specs)
87     return [make_task(seed, f'learn-{i:03}', 'atomic_acquisition', rev, req)
88             for i, (rev, req) in enumerate(specs)]
89
90
91 def evaluation_tasks(seed, prefix='test', size=240):
92     r = random.Random(seed + (3000 if prefix == 'test' else 4000))
93     compositions = [dict(zip(STAGES, values)) for values in itertools.product(*(EFFECTS[s] for s in STAGES))
94                   if sum(a != DEFAULT_GOALS[s] for s, a in zip(STAGES, values)) >= 2]
95     tasks = []
96     for i in range(size):
97         group = ('atomic_new', 'heldout_composition', 'schema_shift', 'new_tool_revision')[i // (size // 4)]
98         rev = 'C' if group == 'new_tool_revision' else r.choice(('A', 'B'))
99         request = r.choice(atomic_requests()) if group == 'atomic_new' else compositions
100         tasks.append(make_task(seed, f'{prefix}-{i:03}', group, rev, request, shifted=group == 'schema_shift'))
101     return tasks
102
103
104 def micro_task(task):
105     """Same public microcase for all methods; two complete order histories."""
106     order = task.roles['order']
107     chosen = sorted({row[order] for row in task.rows})[:2]
108     return Task(task.task_id + ':micro', task.group, task.tool_revision,
109               dict(task.request), dict(task.roles),
110               [dict(row) for row in task.rows if row[order] in chosen])
111
112
113 def observed_rows(task, rows):
114     """Public adapter: bind varying field names to typed logical roles."""
115     return [{role: row[column] for role, column in task.roles.items()} for row in rows]
116
117
118 def transform(rows, stage, effect):
119     """Implementation of the environment's typed anonymous tool actions."""
120     if stage == 'population':
121         return [dict(row) for row in rows if effect == 'all' or row['settled']]
122     if stage == 'revision':
123         if effect == 'all': return [dict(row) for row in rows]
124         ordered = sorted(rows, key=lambda row: row['updated'], reverse=effect == 'latest')
125         seen, result = set(), []
126         for row in ordered:
127             if row['order'] not in seen: result.append(dict(row)); seen.add(row['order'])

```

```

128         return result
129     divisor = {'yuan': 1, 'cent': 100, 'hundred_yuan': .01}[effect]
130     return [dict(row, amount=round(row['amount'] / divisor, 2)) for row in rows]
131
132
133 def canonical(rows):
134     return sorted(rows, key=lambda row: row['row_id'])
135
136
137 class Environment:
138     def __init__(self, seed):
139         rng = random.Random(seed)
140         self.registry = {}
141         for revision in ('A', 'B', 'C'):
142             self.registry[revision] = {}
143             for stage in STAGES:
144                 values = list(EFFECTS[stage]); rng.shuffle(values)
145                 self.registry[revision][stage] = dict(zip(ACTION_IDS[stage], values))
146
147     def expert_program(self, task):
148         # Only demonstration generation and integrity tests call this method.
149         return {s: next(action for action, effect in self.registry[task.tool_revision][s].items()
150                        if effect == task.request[s]) for s in STAGES}
151
152     def execute(self, task, program):
153         rows = observed_rows(task, task.rows)
154         trace, work = [], 0
155         for stage in STAGES:
156             before = rows
157             effect = self.registry[task.tool_revision][stage][program[stage]]
158             rows = transform(before, stage, effect)
159             work += len(before)
160             trace.append({'stage': stage, 'action': program[stage], 'before': before, 'after': rows})
161         work += len(rows) # aggregate / emit
162         artifact = {'total_yuan': round(sum(row['amount'] for row in rows), 2),
163                   'source_rows': sorted(row['row_id'] for row in rows)}
164         return artifact, trace, work
165
166     def independent_goal_rows(self, rows, stage, goal):
167         """Separate reference implementation, not transform()."""
168         if stage == 'population':
169             return [dict(row) for row in rows if goal != 'settled' or bool(row['settled'])]
170         if stage == 'revision':
171             if goal == 'all': return [dict(row) for row in rows]
172             buckets = defaultdict(list)
173             for row in rows: buckets[row['order']].append(row)
174             chooser = min if goal == 'oldest' else max
175             return [dict(chooser(bucket, key=lambda x: x['updated'])) for bucket in buckets.values()]
176         factor = {'yuan': 100, 'cent': 1, 'hundred_yuan': 10000}[goal]
177         return [dict(row, amount=round(row['amount'] * factor / 100, 2)) for row in rows]
178
179     def feedback(self, task, artifact, trace):
180         passed = {}
181         diagnostics = []
182         for assertion, entry in zip(ASSERTIONS, trace):
183             stage = entry['stage']
184             wanted = self.independent_goal_rows(entry['before'], stage, task.request[stage])
185             passed[assertion] = canonical(wanted) == canonical(entry['after'])
186             if not passed[assertion]: diagnostics.append({'assertion': assertion, **ASSERTIONS[assertion]})
187         reference = observed_rows(task, task.rows)
188         for stage in STAGES:
189             reference = self.independent_goal_rows(reference, stage, task.request[stage])
190         expected = {'total_yuan': round(sum(row['amount'] for row in reference), 2),
191                   'source_rows': sorted(row['row_id'] for row in reference)}
192         accepted = (abs(artifact['total_yuan'] - expected['total_yuan']) < 1e-6
193                   and artifact['source_rows'] == expected['source_rows'] and all(passed.values()))
194         # Numeric answer / reference rows / registry are never in returned feedback.
195         return {'accepted': accepted, 'passed': passed, 'diagnostics': diagnostics}
196
197     def submit(self, task, program):
198         artifact, trace, work = self.execute(task, program)
199         return {'artifact': artifact, 'trace': trace, 'work': work,
200               'feedback': self.feedback(task, artifact, trace)}
201
202
203 def demonstrations(environment, seed):
204     corpus = []
205     for revision in ('A', 'B'):
206         for index, request in enumerate(atomic_requests()):
207             for repeat in range(2):
208                 task = make_task(seed, f'source={revision}-{index}-{repeat}',

```

```

209         'external_atomic', revision, request)
210     program = environment.expert_program(task)
211     result = environment.submit(task, program)
212     assert result['feedback']['accepted']
213     corpus.append({'demo_id': task.task_id, 'task': asdict(task),
214                  'program': program, 'trace': result['trace'],
215                  'accepted': True})
216     random.Random(seed + 2000).shuffle(corpus)
217     return corpus
218
219
220 def infer_effect(entry):
221     """Classify observed state changes. No registry or supplied correct-effect tag."""
222     before, after = entry['before'], entry['after']
223     before_ids = {row['row_id'] for row in before}; after_ids = {row['row_id'] for row in after}
224     if entry['stage'] == 'population':
225         if before_ids == after_ids: return 'all'
226         if after_ids == {row['row_id'] for row in before if row['settled']}: return 'settled'
227     elif entry['stage'] == 'revision':
228         if before_ids == after_ids: return 'all'
229         groups = defaultdict(list)
230         for row in before: groups[row['order']].append(row)
231         for label, choose in (('oldest', min), ('latest', max)):
232             chosen = {choose(group, key=lambda x: x['updated'])['row_id'] for group in groups.values()}
233             if chosen == after_ids: return label
234     else:
235         originals = {row['row_id']: row['amount'] for row in before}
236         ratios = {round(row['amount'] / originals[row['row_id']], 8) for row in after}
237         if len(ratios) == 1:
238             return {1: 'yuan', .01: 'cent', 100: 'hundred_yuan'}.get(next(iter(ratios)))
239     return None
240
241
242 def diagnosed_goals(task, feedback):
243     """Symbolic explanation -> public producer scope -> requested postcondition."""
244     result = []
245     for diagnostic in feedback['diagnostics']:
246         stage = LINEAGE[diagnostic['path']]
247         result.append({'assertion': diagnostic['assertion'], 'stage': stage,
248                      'goal': task.request[stage], 'explanation': diagnostic['explanation']})
249     return result
250
251
252 def failed_count(feedback):
253     return sum(not value for value in feedback['passed'].values())
254
255
256 class Controller:
257     def __init__(self, method, corpus):
258         self.method = method
259         self.corpus = copy.deepcopy(corpus)
260         self.templates = []
261         self.fragments = []
262         for demo in corpus:
263             template = {'revision': demo['task']['tool_revision'], 'request': demo['task']['request'],
264                        'program': demo['program'], 'demo_id': demo['demo_id']}
265             if not any(t['revision'] == template['revision'] and t['request'] == template['request'] for t in self.templates):
266                 self.templates.append(template)
267             for entry in demo['trace']:
268                 effect = infer_effect(entry)
269                 if effect is None: continue
270                 fragment = {'revision': demo['task']['tool_revision'], 'stage': entry['stage'],
271                           'effect': effect, 'action': entry['action'], 'demo_id': demo['demo_id']}
272                 if not any(all(f[k] == fragment[k] for k in ('revision', 'stage', 'effect', 'action')) for f in self.fragments):
273                     self.fragments.append(fragment)
274             self.evidence = {}
275             self.workflows = {}
276
277     def state_hash(self):
278         return hashlib.sha256(json.dumps(self.__dict__, sort_keys=True).encode()).hexdigest()
279
280     def key(self, task, stage):
281         return '|'.join((task.tool_revision, stage, task.request[stage]))
282
283     def whole_key(self, task):
284         return '|'.join([task.tool_revision] + [task.request[s] for s in STAGES])
285
286     def candidate_evidence(self, task, stage, action):
287         return self.evidence.get(self.key(task, stage), {}).get(action, {'success': 0, 'failure': 0,

```

```

288                                                                 'accepted_tasks': [], 'reward': 0))
289
290 def incentive_score(self, record):
291     return (record['success'] + 1) / (record['success'] + record['failure'] + 2)
292
293 def initial_program(self, task):
294     program = dict(DEFAULT_PROGRAM)
295     if self.method == 'pi_no_incentive': return program
296     if self.method == 'trajectory_replay':
297         record = self.workflows.get(self.whole_key(task))
298         if record and len(record['tasks']) >= 2: return dict(record['program'])
299     return program
300     for stage in STAGES:
301         options = self.evidence.get(self.key(task, stage), {})
302         eligible = [(action, record) for action, record in options.items()
303                     if len(record['accepted_tasks']) >= 2
304                     and (self.incentive_score(record) >= .7 if self.method != 'compositional_repair' else True)]
305         if eligible:
306             program[stage] = max(eligible, key=lambda ar: (self.incentive_score(ar[1]), ar[1]['reward']))[0]
307     return program
308
309 def ranked_templates(self, task, stage=None, goal=None):
310     def score(template):
311         similarity = sum(template['request'][s] == task.request[s] for s in STAGES)
312         compatible = template['revision'] == task.tool_revision
313         goal_match = stage is not None and template['request'][stage] == goal
314         return (4 * goal_match + 2 * compatible + similarity, template['demo_id'])
315     return sorted(self.templates, key=score, reverse=True)
316
317 def local_candidates(self, task, stage, goal):
318     if self.method == 'self_trial_only': return list(ACTION_IDS[stage])
319     if self.method == 'pi_no_goal':
320         actions = [template['program'][stage] for template in self.ranked_templates(task)]
321     else:
322         candidates = [f for f in self.fragments if f['stage'] == stage and f['effect'] == goal]
323         candidates.sort(key=lambda f: (f['revision'] == task.tool_revision,
324                                       self.incentive_score(self.candidate_evidence(task, stage, f['action'])),
325                                       if self.method != 'compositional_repair' else 0,
326                                       f['demo_id']), reverse=True)
327         actions = [f['action'] for f in candidates]
328     if self.method != 'pi_no_incentive':
329         known = sorted(self.evidence.get(self.key(task, stage), {}).items(),
330                       key=lambda ar: self.incentive_score(ar[1]), reverse=True)
331         actions = [action for action, record in known if record['success'] > 0] + actions
332     return list(dict.fromkeys(actions + list(ACTION_IDS[stage])))
333
334 def reinforce(self, task, updates, final_program, final_feedback):
335     if self.method == 'pi_no_incentive': return
336     if self.method == 'trajectory_replay':
337         if final_feedback['accepted']:
338             key = self.whole_key(task)
339             record = self.workflows.setdefault(key, {'program': dict(final_program), 'tasks': []})
340             if task.task_id not in record['tasks']: record['tasks'].append(task.task_id)
341         return
342     for update in updates:
343         stage, action = update['stage'], update['action']
344         record = self.evidence.setdefault(self.key(task, stage), {}).setdefault(action,
345                                       {'success': 0, 'failure': 0, 'accepted_tasks': [], 'reward': 0})
346         record['success' if update['passed'] else 'failure'] += 1
347         record['reward'] += update['reward']
348     if final_feedback['accepted']:
349         for stage, action in final_program.items():
350             record = self.evidence.setdefault(self.key(task, stage), {}).setdefault(action,
351                                       {'success': 0, 'failure': 0, 'accepted_tasks': [], 'reward': 0})
352             record['success'] += 1
353             record['reward'] += 1
354             if task.task_id not in record['accepted_tasks']: record['accepted_tasks'].append(task.task_id)
355
356
357 def episode(controller, task, submit, budget=3, learn=False, detailed=False):
358     program = controller.initial_program(task)
359     first_program = dict(program)
360     first = submit(task, program)
361     current_feedback = first['feedback']
362     initial_feedback = copy.deepcopy(current_feedback)
363     full_work, probe_work, trials, events, updates = first['work'], 0, 0, [], []
364     full_submissions = 1
365     regression_commits = 0
366     goals_before = diagnosed_goals(task, current_feedback)
367     micro = micro_task(task)
368     seen = set()

```

```

369 if not current_feedback['accepted'] and controller.method == 'pi_no_trial':
370     for diagnosis in goals_before:
371         stage, goal = diagnosis['stage'], diagnosis['goal']
372         program[stage] = controller.local_candidates(task, stage, goal)[0]
373         events.append({'scope': [stage], 'goal': goal, 'action': program[stage],
374                        'trial': False, 'reason': diagnosis['explanation']})
375 elif not current_feedback['accepted']:
376     if controller.method == 'trajectory_replay':
377         best_failures = failed_count(current_feedback)
378         for template in controller.ranked_templates(task):
379             if trials >= budget: break
380             proposal = dict(template['program'])
381             signature = json.dumps(proposal, sort_keys=True)
382             if signature in seen or proposal == program: continue
383             seen.add(signature)
384             result = submit(micro, proposal); trials += 1; probe_work += result['work']
385             feedback = result['feedback']
386             committed = failed_count(feedback) < best_failures
387             regressions = sum(value and not feedback['passed'][assertion]
388                               for assertion, value in current_feedback['passed'].items())
389             if committed:
390                 program = proposal; current_feedback = feedback; best_failures = failed_count(feedback)
391                 regression_commits += regressions
392             events.append({'scope': list(STAGES), 'goal': 'complete_request',
393                           'candidate_source': template['demo_id'], 'program': proposal,
394                           'probe_passed': feedback['accepted'], 'committed': committed,
395                           'regressions': regressions if committed else 0,
396                           'failed_constraints': failed_count(feedback)})
397             if feedback['accepted']: break
398     else:
399         while trials < budget:
400             diagnoses = diagnosed_goals(task, current_feedback)
401             if not diagnoses: break
402             diagnosis = diagnoses[0]
403             stage, goal = diagnosis['stage'], diagnosis['goal']
404             proposals = []
405             if controller.method == 'pi_no_localization':
406                 for template in controller.ranked_templates(task, stage, goal):
407                     proposals.append((dict(template['program']), template['demo_id']))
408             else:
409                 for action in controller.local_candidates(task, stage, goal):
410                     proposal = dict(program); proposal[stage] = action
411                     proposals.append((proposal, 'effect_fragment_or_own_trial'))
412             proposed = next(((proposal, source) for proposal, source in proposals
413                             if proposal != program and
414                             (stage, json.dumps(proposal, sort_keys=True)) not in seen), None)
415             if proposed is None: break
416             proposal, source = proposed
417             seen.add((stage, json.dumps(proposal, sort_keys=True)))
418             result = submit(micro, proposal); trials += 1; probe_work += result['work']
419             feedback = result['feedback']
420             passed = feedback['passed'][diagnosis['assertion']]
421             reward = failed_count(current_feedback) - failed_count(feedback) - .01 * result['work'] / max(1, len
422 (micro.rows))
423             # A localized pass is tentative. Full-task acceptance is required
424             # for proactive future reuse of this conditioned action.
425             updates.append({'stage': stage, 'action': proposal[stage], 'passed': passed,
426                            'reward': reward if passed else min(0, reward)})
427             regressions = sum(value and not feedback['passed'][assertion]
428                               for assertion, value in current_feedback['passed'].items())
429             if passed:
430                 program = proposal; current_feedback = feedback
431                 regression_commits += regressions
432             events.append({'scope': list(STAGES) if controller.method == 'pi_no_localization' else [stage],
433                           'goal': goal, 'assertion': diagnosis['assertion'], 'reason': diagnosis['explanation']
434                           },
435                           'candidate_source': source, 'program': proposal, 'probe_passed': passed,
436                           'committed': passed, 'regressions': regressions if passed else 0,
437                           'incentive': reward if passed else min(0, reward),
438                           'failed_constraints': failed_count(feedback)})
439 if not first['feedback']['accepted']:
440     final = submit(task, program); full_submissions += 1; full_work += final['work']
441 else:
442     final = first
443     final_feedback = final['feedback']
444     if learn: controller.reinforce(task, updates, program, final_feedback)
445     regression = any(value and not final_feedback['passed'][assertion]
446                     for assertion, value in initial_feedback['passed'].items())
447     assert full_submissions <= 2 and trials <= budget
448     result = {'task_id': task.task_id, 'group': task.group, 'tool_revision': task.tool_revision,
449              'success': int(final_feedback['accepted']), 'first_success': int(first['feedback']['accepted']),

```

```

448         'initial_failed_constraints': failed_count(initial_feedback),
449         'final_failed_constraints': failed_count(final_feedback),
450         'small_trials': trials, 'full_submissions': full_submissions,
451         'row_stage_visits': full_work + probe_work, 'probe_row_stage_visits': probe_work,
452         'final_collateral_regression': int(regression),
453         'collateral_regression_commits': regression_commits,
454         'probe_failures': sum(e.get('probe_passed') is False for e in events),
455         'accepted_incentive': sum(e.get('incentive', 0) for e in events if e.get('committed')),
456         'initial_program': first_program, 'final_program': dict(program), 'events': events)
457     if detailed:
458         result.update(task=asdict(task), initial_response=first, final_response=final)
459     return result
460
461
462 def frozen_batch(environment, controller, tasks, budget, seed, mode):
463     before = controller.state_hash()
464     rows = []
465     for task in tasks:
466         row = episode(controller, task, environment.submit, budget=budget)
467         row.update(seed=seed, method=controller.method, mode=mode, trial_budget=budget)
468         rows.append(row)
469     assert controller.state_hash() == before, 'Persistent state changed during frozen evaluation'
470     return rows
471
472
473 def summarize(rows, keys):
474     grouped = defaultdict(list)
475     for row in rows: grouped[tuple(row[k] for k in keys)].append(row)
476     metrics = ['success', 'first_success', 'initial_failed_constraints', 'final_failed_constraints',
477               'small_trials', 'full_submissions', 'row_stage_visits', 'probe_row_stage_visits',
478               'final_collateral_regression', 'collateral_regression_commits', 'probe_failures', 'accepted_incentive']
479     result = []
480     for group, entries in sorted(grouped.items()):
481         item = dict(zip(keys, group)); item['episodes'] = len(entries)
482         for metric in metrics: item[metric] = statistics.mean(r[metric] for r in entries)
483         result.append(item)
484     return result
485
486
487 def write_csv(path, rows):
488     with path.open('w', encoding='utf-8-sig', newline='') as f:
489         writer = csv.DictWriter(f, list(rows[0])); writer.writeheader(); writer.writerows(rows)
490
491
492 def experiment():
493     OUT.mkdir(parents=True, exist_ok=True)
494     start = time.perf_counter()
495     final, acquisition, curves, examples, corpora, task_archive, snapshots, checks = [], [], [], [], [], [], [], []
496     for seed in SEEDS:
497         environment = Environment(seed)
498         corpus = demonstrations(environment, seed)
499         corpora.extend(dict(seed=seed, **entry) for entry in corpus)
500         train = acquisition_tasks(seed); tests = evaluation_tasks(seed); probes = evaluation_tasks(seed, 'probe',
501         24)
502         task_archive.extend({'seed': seed, **asdict(task)} for task in train + tests + probes)
503         for method in METHODS:
504             controller = Controller(method, corpus)
505             final.extend(frozen_batch(environment, controller, tests, 3, seed, 'cold'))
506             for checkpoint in (0, 24, 48, 96):
507                 if checkpoint:
508                     previous = {24: 0, 48: 24, 96: 48}[checkpoint]
509                     for task in train[previous:checkpoint]:
510                         row = episode(controller, task, environment.submit, budget=3, learn=True)
511                         row.update(seed=seed, method=method, mode='acquisition', trial_budget=3)
512                         acquisition.append(row)
513                     rows = frozen_batch(environment, controller, probes, 3, seed, f'checkpoint_{checkpoint}')
514                     for row in rows: row['checkpoint'] = checkpoint
515                     curves.extend(rows)
516                 for budget in (1, 3, 6):
517                     final.extend(frozen_batch(environment, controller, tests, budget, seed, 'trained'))
518             checks.append({'seed': seed, 'method': method, 'frozen_evaluation': True,
519                           'condition_keys': len(controller.evidence), 'whole_workflows': len(controller.workflows)})
520     })
521     if method == 'pi_full':
522         snapshots.append({'seed': seed, 'evidence': controller.evidence,
523                           'fragments': controller.fragments, 'state_hash': controller.state_hash()})
524         reset = Controller(method, corpus)
525         reset_rows = frozen_batch(environment, reset, tests, 3, seed, 'reset_verified_state')
526         final.extend(reset_rows)
527     if seed == SEEDS[0] and method in ('pi_full', 'trajectory_replay', 'pi_no_localization',

```

```

526                                     'pi_no_trial', 'pi_no_goal'):
527     fresh = Controller(method, corpus)
528     for index in (60, 61, 62, 120, 180, 181, 182):
529         row = episode(fresh, tests[index], environment.submit, budget=3, detailed=True)
530         row.update(seed=seed, method=method, mode='cold_trace_example', trial_budget=3)
531         examples.append(row)
532     print(f'seed {seed}: completed', flush=True)
533     write_csv(OUT/'summary.csv', summarize(final, ['mode', 'trial_budget', 'method', 'group']))
534     write_csv(OUT/'per_seed_summary.csv', summarize(final, ['seed', 'mode', 'trial_budget', 'method', 'group']))
535     write_csv(OUT/'learning_curves.csv', summarize(curves, ['checkpoint', 'method', 'group']))
536     write_csv(OUT/'acquisition_summary.csv', summarize(acquisition, ['seed', 'method']))
537     for filename, records in [('episodes.jsonl', final), ('acquisition_episodes.jsonl', acquisition),
538                             ('trace_examples.jsonl', examples), ('demonstrations.jsonl', corpora),
539                             ('tasks.jsonl', task_archive)]:
540         with (OUT/filename).open('w', encoding='utf-8') as f:
541             for record in records: f.write(json.dumps(record, ensure_ascii=False) + '\n')
542     (OUT/'learned_incentives.json').write_text(json.dumps(snapshots, indent=2), encoding='utf-8')
543     metrics = {'status': 'executed_cpu_correction_loop', 'seeds': SEEDS, 'methods': METHODS,
544              'trial_budgets': [1, 3, 6], 'full_submission_limit': 2,
545              'external_demonstrations_per_seed': 24, 'acquisition_tasks_per_seed': 96,
546              'final_tasks_per_seed': 240, 'main_final_episodes': len(final),
547              'acquisition_episodes': len(acquisition), 'checkpoint_episodes': len(curves),
548              'detailed_example_executions': len(examples),
549              'total_episodes': len(final) + len(acquisition) + len(curves) + len(examples),
550              'elapsed_seconds': time.perf_counter()-start,
551              'llm_calls': 0, 'trained_neural_parameters': 0,
552              'state_checks': checks,
553              'script_sha256': hashlib.sha256(Path(__file__).read_bytes()).hexdigest(),
554              'limitations': ['finite typed tools and symbolic goal parsing',
555                             'structured instrumented assertion explanations available equally to all policies',
556                             'schema adapter and executable composition grammar supplied',
557                             'small fixtures identify this finite operator vocabulary; not arbitrary-program safety',
558                             'reinforcement updates counts and preferences, not neural parameters',
559                             'not LLM baseline reproductions or token-budget measurements']}
560     (OUT/'metrics.json').write_text(json.dumps(metrics, indent=2), encoding='utf-8')
561     print(json.dumps({k:v for k,v in metrics.items() if k not in ('state_checks', 'limitations')}, indent=2))
562
563
564 if __name__ == '__main__':
565     experiment()
    
```

A.2. Exhaustive Integrity Checks

```

1  """Integrity checks performed before inspecting controller scores."""
2  import copy
3  import itertools
4  import json
5  from collections import Counter
6  from dataclasses import fields
7  import run_correction as p
8
9
10 def verify():
11     task_count = program_count = 0
12     for seed in p.SEEDS:
13         env = p.Environment(seed)
14         train, tests, probes = p.acquisition_tasks(seed), p.evaluation_tasks(seed), p.evaluation_tasks(seed, 'probe'
15         , 24)
16         corpus = p.demonstrations(env, seed)
17         assert len(train) == 96 and len(corpus) == 24
18         assert set(Counter((t.tool_revision, tuple(t.request[s] for s in p.STAGES)) for t in train).values()) == {8}
19         assert set(Counter(t.group for t in tests).values()) == {60}
20         assert all(sum(t.request[s] != p.DEFAULT_GOALS[s] for s in p.STAGES) <= 1 for t in train)
21         assert all(sum(d['task']['request'][s] != p.DEFAULT_GOALS[s] for s in p.STAGES) <= 1 for d in corpus)
22         sets = [{t.task_id for t in train}, {t.task_id for t in tests}, {t.task_id for t in probes},
23                 {d['demo_id'] for d in corpus}]
24         assert sum(map(len, sets)) == len(set.union(*sets))
25         for demo in corpus:
26             for entry in demo['trace']:
27                 inferred = p.infer_effect(entry)
28                 actual = env.registry[demo['task']['tool_revision']][entry['stage']][entry['action']]
29                 assert inferred == actual
30         for task in train + tests + probes:
31             expert = env.expert_program(task)
32             golden = env.submit(task, expert)
33             assert golden['feedback']['accepted']
34             bad = dict(golden['artifact'], total_yuan=golden['artifact']['total_yuan'] + 1)
35             assert not env.feedback(task, bad, golden['trace']['accepted'])
36             bad = dict(golden['artifact'], source_rows=golden['artifact']['source_rows'][:-1])
37             assert not env.feedback(task, bad, golden['trace']['accepted'])
38             accepted_programs = []
39             for actions in itertools.product(*[p.ACTION_IDS[s] for s in p.STAGES]):
40                 program = dict(zip(p.STAGES, actions))
41                 full = env.submit(task, program)
42                 micro = env.submit(p.micro_task(task), program)
43                 # In this deliberately finite vocabulary, the shared fixture
44                 # must separate actions exactly. This is NOT a claim that small
45                 # tests certify arbitrary programs.
46                 assert full['feedback']['passed'] == micro['feedback']['passed'], (seed, task.task_id, program)
47                 if full['feedback']['accepted']: accepted_programs.append(program)
48                 assert set(full['feedback']) == {'accepted', 'passed', 'diagnostics'}
49                 assert all('action' not in d and 'stage' not in d for d in full['feedback']['diagnostics'])
50                 program_count += 1
51             assert accepted_programs == [expert]
52             task_count += 1
53         for method in p.METHODS:
54             controller = p.Controller(method, corpus)
55             assert len(controller.fragments) == 16
56             original = controller.state_hash()
57             results = p.frozen_batch(env, controller, tests[:8] + tests[-8:], 3, seed, 'integrity')
58             assert controller.state_hash() == original
59             assert all(r['small_trials'] <= 3 and r['full_submissions'] <= 2 for r in results)
60             for result in results:
61                 if method not in ('pi_no_localization', 'trajectory_replay'):
62                     assert all(len(e['scope']) == 1 for e in result['events'])
63                     if method == 'pi_no_trial': assert result['small_trials'] == 0
64             for task in train[:8]:
65                 p.episode(controller, task, env.submit, 3, learn=True)
66             if method == 'pi_no_incentive':
67                 assert controller.evidence == {} and controller.workflows == {}
68             # Frozen evaluation does not reinforce skills; acquisition does.
69             learner = p.Controller('pi_full', corpus)
70             before = learner.state_hash()
71             for task in train:
72                 p.episode(learner, task, env.submit, 3, learn=True)
73             assert learner.state_hash() != before
74             assert learner.evidence
75         assert {f.name for f in fields(p.Task)} == {
76             'task_id', 'group', 'tool_revision', 'request', 'roles', 'rows'}
77         result = {'status': 'passed', 'generated_tasks_checked': task_count,
78                 'full_micro_program_comparisons': program_count,
79                 'seeds': p.SEEDS,
    
```

```
79         'checks': ['independent evaluator and executor agreement',
80                     'wrong aggregate/provenance rejected',
81                     'unique identifying program for each task',
82                     'micro fixture agrees with full assertions for every anonymous program',
83                     'demo fragment effects inferred from actual state changes',
84                     'held-out multi-goal compositions',
85                     'disjoint task/example identifiers',
86                     'same budgets and public feedback',
87                     'feedback omits correct action and hidden fault stage',
88                     'one-stage patch scope except explicit no-scope and trajectory controls',
89                     'frozen evaluation hashes',
90                     'no-incentive control retains no verified action evidence']]
91     p.OUT.mkdir(parents=True, exist_ok=True)
92     (p.OUT/'integrity_checks.json').write_text(json.dumps(result, indent=2), encoding='utf-8')
93     print(json.dumps(result, indent=2))
94
95
96 if __name__ == '__main__':
97     verify()
```

A.3. Exploratory Online Adaptation

```

1  """Exploratory online adaptation, declared after the primary frozen pilot.
2
3  Frozen unseen-revision evaluation cannot measure learning across new-revision
4  tasks. Here we allow 24 separate C acquisition tasks, then freeze again at
5  0/6/12/24 exposures. Same original seeds, K=3, operators and final C tasks.
6  No score-dependent change to the primary experiment.
7  """
8  import itertools
9  import json
10 import random
11 import time
12 import run_correction as p
13
14
15 def adaptation_tasks(seed):
16     requests = [dict(zip(p.STAGES, choices))
17                 for choices in itertools.product(*(p.EFFECTS[s] for s in p.STAGES))
18                 if sum(choice != p.DEFAULT_GOALS[s] for s, choice in zip(p.STAGES, choices)) >= 2]
19     assert len(requests) == 12
20     requests = requests * 2
21     random.Random(seed + 5000).shuffle(requests)
22     return [p.make_task(seed, f'adapt-C-{i:03}', 'online_C_acquisition', 'C', request)
23             for i, request in enumerate(requests)]
24
25
26 def experiment():
27     started = time.perf_counter()
28     results, acquisition, archived_tasks = [], [], []
29     methods = ['pi_full', 'pi_no_incentive', 'self_trial_only', 'compositional_repair']
30     warmup_episodes = 0
31     for seed in p.SEEDS:
32         env = p.Environment(seed)
33         corpus = p.demonstrations(env, seed)
34         stream = adaptation_tasks(seed)
35         tests = [task for task in p.evaluation_tasks(seed) if task.group == 'new_tool_revision']
36         assert not {t.task_id for t in stream}.intersection({t.task_id for t in tests})
37         archived_tasks.extend({'seed': seed, **p.asdict(task)} for task in stream)
38         for method in methods:
39             controller = p.Controller(method, corpus)
40             for task in p.acquisition_tasks(seed):
41                 p.episode(controller, task, env.submit, budget=3, learn=True)
42                 warmup_episodes += 1
43             previous = 0
44             for checkpoint in (0, 6, 12, 24):
45                 for task in stream[previous:checkpoint]:
46                     row = p.episode(controller, task, env.submit, budget=3, learn=True)
47                     row.update(seed=seed, method=method, mode='online_C_acquisition', trial_budget=3)
48                     acquisition.append(row)
49                     rows = p.frozen_batch(env, controller, tests, 3, seed, 'after_online_C')
50                     for row in rows: row['checkpoint'] = checkpoint
51                     results.extend(rows)
52                     previous = checkpoint
53     p.write_csv(p.OUT/'online_adaptation_summary.csv', p.summarize(results, ['checkpoint', 'method', 'group']))
54     p.write_csv(p.OUT/'online_adaptation_per_seed.csv', p.summarize(results, ['seed', 'checkpoint', 'method', 'group']
55     ))
56     p.write_csv(p.OUT/'online_adaptation_acquisition.csv', p.summarize(acquisition, ['seed', 'method']))
57     for filename, records in [(p.OUT/'online_adaptation_episodes.jsonl', results),
58                               (p.OUT/'online_adaptation_tasks.jsonl', archived_tasks),
59                               (p.OUT/'online_adaptation_learning.jsonl', acquisition)]:
60         with (p.OUT/filename).open('w', encoding='utf-8') as f:
61             for row in records: f.write(json.dumps(row) + '\n')
62     metrics = {'status': 'executed_exploratory_online_adaptation',
63               'declared_after_primary_scores': True,
64               'primary_protocol_unchanged': True,
65               'new_C_acquisition_tasks_per_seed': 24, 'trial_budget': 3,
66               'checkpoints': [0, 6, 12, 24], 'methods': methods,
67               'warmup_episodes': warmup_episodes, 'adaptation_episodes': len(acquisition),
68               'frozen_evaluation_episodes': len(results),
69               'total_episodes': warmup_episodes + len(acquisition) + len(results),
70               'elapsed_seconds': time.perf_counter() - started,
71               'llm_calls': 0,
72               'limitation': 'After C acquisition, this is retention in a learned C environment, not zero-shot
73               transfer to an unseen tool revision.'}
74     (p.OUT/'online_adaptation_metrics.json').write_text(json.dumps(metrics, indent=2), encoding='utf-8')
75     print(json.dumps(metrics, indent=2))
76
77 if __name__ == '__main__':
78     experiment()

```