CONSEQHARNESS: MEMORY-CONDITIONED SEQUENTIAL OPTIMIZATION OF AGENT HARNESSES

Anonymous authors
Paper under double-blind review

ABSTRACT

As large language models (LLMs) increasingly rely on tools, routing, persistent state, and verification for long-horizon execution, agent performance depends not only on the model itself but also on the external harness that organizes its execution. Existing optimizers use prior code, scores, and traces to guide new proposals, but typically treat such experience as proposal context rather than reusable, validated optimization state. We introduce the verified harness transition as a reusable optimization primitive that binds a motivating failure to a typed and scoped edit, its measured gain, validation evidence, and non-target effects. This turns sequential harness optimization into a memory-augmented process in which previously validated changes can explicitly condition future decisions. We formalize this view as the *Harness-Reflected Decision Process* and instantiate it with Memory-***Con***ditioned ***Seq***uential Optimization of Agent ***Harness***es (ConSeqHarness), a constrained sequential optimizer that retrieves compatible verified transitions, proposes bounded structural edits, and atomically updates both the harness and transition memory only after structural, runtime, improvement, and regression checks. Across three target models and five public agent benchmarks, ConSeqHarness ranks first in all 15 settings and improves macro-average accuracy over the strongest baseline by 2.73–3.43 points under the same candidate-attempt budget. Controlled ablations further show that retaining accepted experience and representing it as structured transitions both contribute to the gains, while unverified memory and unconstrained editing substantially increase regressions and runtime failures. *The source code is available at* `https://anonymous.4open.science/r/ConSeqHarness-579E`.

1 INTRODUCTION

Large language models (LLMs) increasingly operate through tools, persistent state, and multi-step procedures rather than isolated responses (Ye et al., 2025; Long et al., 2026). Errors can therefore arise outside the model call: a router may select the wrong branch, a parser may discard a field, or a terminal node may map a correct intermediate result to an invalid answer. Such failures are systematic because the same executable path is reused across task instances. Performance consequently depends on the agent harness, including the calls, data dependencies, control flow, state updates, and verification logic (Zhai et al., 2025; Lee et al., 2026). Improving this layer complements model training and prompt optimization. Unlike test-time scaling within a fixed call pattern (Wang et al., 2026b; Yu et al., 2026), harness optimization changes the computation executed by future task instances.

Harness optimization is naturally sequential: each accepted edit changes both the executable state and the remaining failures seen by later rounds. This creates three coupled difficulties. *Credit assignment* requires separating the local change responsible for improvement from incidental differences elsewhere in a candidate. *Applicability* requires deciding whether an earlier repair transfers to the current schema, branch, and failure type. *Safe accumulation* requires preventing rejected, incomplete, or regressive proposals from reintroducing errors that previous rounds removed. These difficulties interact: an incorrectly attributed success may be retrieved in an incompatible context and then contaminate future revisions. Useful memory must therefore record not only what was tried, but where it applied, what changed, and which checks it passed.

Recent methods already exploit history and validation: workflow optimizers search candidate programs (Wang et al., 2025; Gao et al., 2025), Meta-Harness conditions revisions on prior code and

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

feedback (Lee et al., 2026), and Self-Harness promotes tested failure-driven changes (Zhang et al., 2026a). Our claim is not that these methods are memoryless. Rather, a chronological log records events and a full candidate records program state, but neither necessarily exposes the precondition, local effect boundary, and non-target behavior needed for later reuse. We introduce the *validation-certified harness transition*: a record that binds failure evidence and provenance to a typed, scoped edit and its structural, runtime, gain, and regression outcomes. It becomes readable only when the corresponding harness is accepted, so memory cannot claim the effect of a candidate while the executable state remains unchanged. Compatibility filtering then restricts reuse by task mode, schema, output contract, failure type, and target boundary. The retrieved object is therefore an empirically grounded edit prior rather than an unconstrained textual suggestion.

CONSEQHARNESS represents a harness as a type-constrained workflow graph and restricts proposals to scoped node, branch, or terminal edits. Each round diagnoses failures, retrieves compatible accepted transitions, proposes one typed edit, and applies a risk-progressive cascade. Static checks reject malformed graphs before execution, smoke tests establish path executability, diagnostic evaluation requires measurable gain, and a guard set limits regressions on current successes and previously repaired cases. Only a fully validated candidate atomically updates both the harness and memory, while rejection preserves both accepted states. The matched *Flat Accepted History* condition retains the same accepted evidence, editor, retrieval budget, operators, and gate but removes typed transition fields, isolating representation from the broader benefit of successful history. At deployment, the editor and memory are removed and only the compiled final harness executes.

We formalize the optimizer over the augmented state $\bar{s}_t = (s_t, M_t)$, coupling the current harness and evidence with accepted transition memory. The formulation makes history dependence and the atomic read–write interface explicit. Appendix A establishes that under the operational diagnostic/guard partition, every accepted update increases the number of correct training cases while static validation preserves graph feasibility. These optimization-side invariants explain accepted-state evolution under this protocol without guaranteeing distribution-free or monotonic held-out improvement.

Across five public benchmarks and three target models, ConSeqHarness leads all 15 reported settings under a shared 18-attempt budget and improves macro-average accuracy over the strongest reported baseline by 2.73–3.43 points. Flat Accepted History is 1.67–1.94 points worse on the two ablated models, and removing memory causes larger drops. Reliability metrics follow the same ordering. Multi-manifest, retention, and deployment audits further test sampling sensitivity, persistence of accepted repairs, and whether gains arise from additional test-time model calls or output tokens.

In summary, our contributions are as follows:

- We formulate sequential harness optimization using a validation-certified transition record that exposes applicability, a typed edit contract, and observed target and non-target effects.
- We propose ConSeqHarness, combining compatibility-filtered transition retrieval, bounded graph edits, and an atomic validation gate that advances executable and memory states.
- We demonstrate gains across five benchmarks and three models under a shared proposal budget, improving macro-average accuracy by 2.73–3.43 points over the strongest baseline, with ablations and audits supporting robustness, retention, and deployment efficiency.

2 RELATED WORK

Textual optimization, workflow search, and harness optimization. Prompt-program optimizers improve instructions, demonstrations, or textual variables while leaving most of the surrounding interface fixed. MIPRO searches instructions and demonstrations (Opsahl-Ong et al., 2024), TextGrad propagates textual feedback (Yuksekgonul et al., 2024), and GEPA evolves prompts through reflection (Agrawal et al., 2026). ConSeqHarness instead allows changes to deterministic code, typed field propagation, routing, tools, and terminal mappings, so candidates must remain executable graphs rather than only high-scoring text. Workflow methods including AFlow (Zhang et al., 2025a), AutoFlow (Li et al., 2024), Flow (Niu et al., 2025), DyFlow (Wang et al., 2026a), FlowReasoner (Gao et al., 2025), and ScoreFlow (Wang et al., 2025) generate or refine executable structures. RobustFlow (Xu et al., 2025) and failure-driven refinement (Zhang et al., 2025b) emphasize robust correction. Closest to our setting, Meta-Harness (Lee et al., 2026) revises code from execution history, Self-Harness (Zhang et al., 2026a) promotes tested local modifications, Natural-Language Agent

Harnesses (Pan et al., 2026) express control logic in language, and Continual Harness (Karten et al., 2026) studies online adaptation. These approaches primarily differ in search space, feedback source, and whether revisions are local or global. our orthogonal concern is the unit of reusable evidence passed between search steps. The Flat Accepted History control removes typed transition structure while holding accepted evidence and optimization machinery fixed.

Reusable experience, test-time scaling, and verification. Trace2Skill (Ni et al., 2026), MUSE-Autoskill (Lin et al., 2026), SkillOpt (Yang et al., 2026), and CoevoSkills (Zhang et al., 2026b) turn trajectories or feedback into reusable textual skills, while Memento-II (Wang, 2025) motivates explicit read–write memory. These methods establish the value of persistence, but a skill is commonly interpreted by the task agent and may remain in deployment context. A ConSeqHarness record is optimizer-side provenance: it conditions graph edits, is admitted only after its executable change passes the full gate, and is absent at deployment. This lifecycle separates optimization experience from the deployed context and ties every readable record to a harness state that actually occurred. Test-time scaling (Zhu et al., 2025b; Shen et al., 2025; Li et al., 2026) instead improves an individual rollout by increasing reasoning, interaction, search, or aggregation compute. Harness optimization amortizes offline search into a changed program, motivating our audit of longest-path LLM-node count and token cap. Finally, validation here is not only a candidate-selection mechanism: structural, runtime, diagnostic, and guard checks jointly determine whether the harness becomes current and whether the transition becomes reusable memory.

3 Problem Formulation

3.1 Harness Representation and Objective

We define an agent harness as an external executable structure that organizes execution units and control flow around a frozen model. We represent it as a type-constrained workflow graph:

$$W = (V, E, C), \tag{1}$$

where V contains executable nodes with typed input/output fields, E denotes data and control dependencies, and C denotes structural constraints such as type consistency, tool signatures, routing validity, and output format requirements. This representation exposes the agent's external execution logic while requiring no access to model parameters or skill internals.

Let D_{diag}, D_{guard}, D_{smoke}, and D_{test} denote diagnostic, guard, smoke-test, and held-out test examples, respectively. The first three are used during optimization to expose failures, detect regressions, and check executability. The goal is to find a final harness W_T that improves performance on unseen examples under validity, regression, and budget constraints:

$$\begin{aligned} \max_{W_T} \quad & \mathbb{E}_{x\sim\mathcal{D}_{\text{task}}}\big[\text{Metric}(W_T, x)\big] \\ \text{s.t.} \quad & \text{StaticPass}(W_T) = 1, \quad \text{RuntimePass}(W_T, D_{\text{smoke}}) = 1, \\ & \text{Regression}(W_T, D_{\text{guard}}) \le \epsilon, \quad \text{Cost}(W_{0:T}) \le B. \end{aligned} \tag{2}$$

Here, $\mathcal{D}_{\text{task}}$ denotes the underlying task distribution, and $\text{Metric}(W_T, D_{\text{test}})$ is reported only as a held-out empirical estimate. Since each execution changes the available evidence and admissible edits, this objective induces a sequential optimization process rather than a one-step update rule.

3.2 Memory-Augmented Sequential State

At round t, executing W_t produces trajectories ξ_t and evidence e_t. Let κ_t contain the round, remaining budget, smoke set, and short-term rejected-candidate deduplication state. We define

$$s_t = (W_t, \xi_t, e_t, D^t_{\text{diag}}, D^t_{\text{guard}}, \kappa_t), \qquad \bar{s}_t = (s_t, M_t), \tag{3}$$

where M_t stores accepted transitions accessible to the editor.

Since identical s_t can arise from histories with different validated transitions, M_t is needed to determine future edits. Under a fixed read and write interface, harness optimization forms a constrained transition process over the augmented state $\bar{s}_t$:

$$p_t \in \mathcal{A}(\bar{s}_t), \qquad \bar{s}_{t+1} \sim P(\cdot \mid \bar{s}_t, p_t), \tag{4}$$

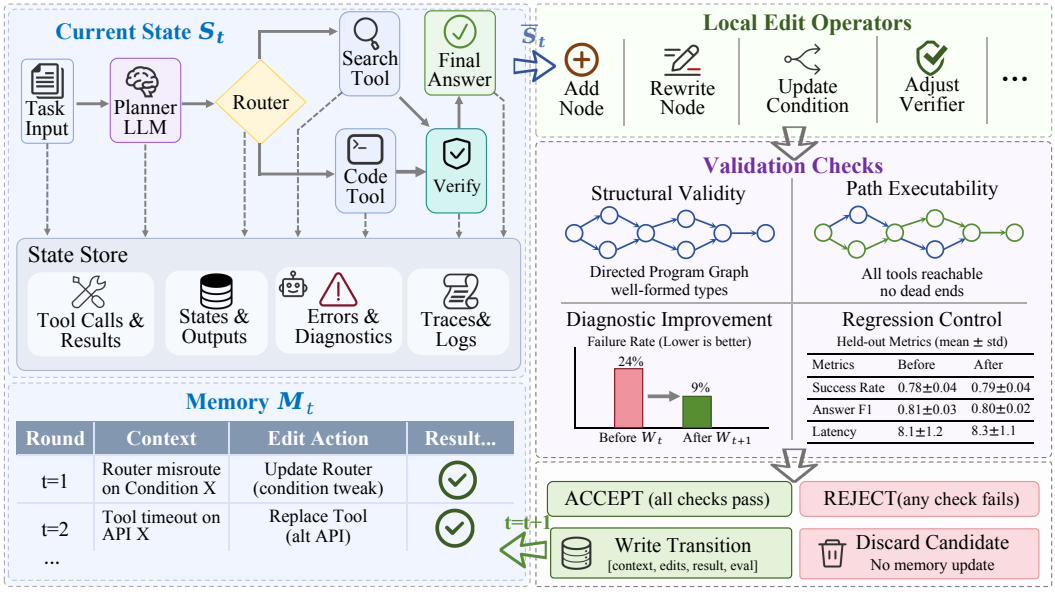


Figure 1: **The ConSeqHarness Sequential Optimization Loop.** The current harness and execution evidence s_t, together with accepted transition memory M_t, form the augmented optimization state $\bar{s}_t$. Failure evidence retrieves compatible validated transitions, and an LLM editor proposes a typed, scoped local edit. The validation cascade either jointly commits the edited harness and its transition record to $\bar{s}_{t+1}$, or preserves the accepted state.

where each action is a typed edit $p = (\text{op}, \text{target}, \text{pre}, \delta, \text{post}, \text{scope})$ in the structurally admissible set $\mathcal{A}(\bar{s}_t)$. Appendix A formalizes closure, feasibility, certified writes, and monotonic training-side progress under the operational gate.

3.3 Memory-Conditioned Policy and Reward

The optimization memory stores evaluated transitions, with each entry $m_i = (\phi_i, p_i, r_i, c_i, g_i)$ recording failure evidence, the applied edit, reward, validation checks, and guard effects.

At round t, the evidence is summarized as $\phi_t = \Phi(e_t)$, and memory retrieval selects a fragment $z \sim \rho(\cdot \mid s_t, M_t)$. An LLM editor with fixed parameters θ induces an editing kernel $K_\theta(p \mid s_t, z)$, yielding the memory-conditioned policy:

$$\pi_\theta(p \mid s_t, M_t) = \sum_z \rho(z \mid s_t, M_t) K_\theta(p \mid s_t, z). \tag{5}$$

After applying p_t, the updated harness is executed and validated, producing reward $r_t = R(s_t, p_t, s_{t+1})$, check record c_t, and guard impact g_t. The memory is then updated by

$$M_{t+1} = U(M_t, \phi_t, p_t, r_t, c_t, g_t). \tag{6}$$

The reward balances diagnostic gain, harness validity, guard preservation, and cost for transition selection, while final performance is reported only on D_{test}.

4 ConSeqHarness Framework

This section instantiates Section 3 as validation-certified transition optimization. As illustrated in Figure 1, typed local edits specify effect scope, and risk-progressive validation determines whether a proposal atomically updates both the executable harness and the memory available to future rounds.

4.1 Harness Graph Specification and Initialization

We instantiate the abstract workflow $W = (V, E, C)$ as a strongly typed directed program graph. Each node $v \in V$ represents an atomic execution unit, with five core roles covering system input,

model invocation, deterministic computation, conditional routing, and terminal output. These roles are implemented through seven canonical node classes detailed in Appendix B.1, with additional support for explicit intermediate decisions, parallel execution, and bounded loops when required. Named field references define the data dependencies E, while C enforces configuration validity, routing completeness, and terminal-schema consistency.

When W_0 is absent, ConSeqHarness compiles a task-specific ReAct-equivalent (Yao et al., 2022) graph into explicit node handles, as detailed in Appendix B.2. The smoke-checked W_0 initializes methods that accept an executable harness and is distinct from the fixed ReAct baseline in Table 1.

4.2 Hierarchical Action Space

To prevent unconstrained workflow rewriting, ConSeqHarness restricts the admissible edit set $\mathcal{A}(\bar{s}_t)$ to three structural tiers ordered by effect radius. The editor conditions on the failure summary ϕ_t and a transition fragment z_t retrieved from compatible validated transitions in M_t. ***Level 1 Intra-node Updates*** modify node-local instructions, prompt templates, deterministic code, tool configurations, output mappings, or hyperparameters without changing graph topology. ***Level 2 Local Branching*** adds conditional routing edges and remediation subgraphs near the faulty node while preserving routing completeness and default fallbacks. ***Level 3 Terminal Reflection*** appends evaluation loops or consensus nodes before the terminal output mapping to stabilize decision schemas. Each operator declares typed preconditions and postconditions that constrain free-form LLM proposals to bounded graph edits informed by the retrieved transition prior. Detailed transaction serialization and retrieval logic are documented in Appendix B.3. The full operator set is defined in Appendix B.4.

4.3 Transition Read–Write Interface

Each accepted record separates three roles that a chronological optimization log conflates. ***Applicability fields*** identify task mode, workflow schema, output contract, failure type, target node, and trigger boundary. The ***edit contract*** stores $(\mathrm{op}, \mathrm{target}, \mathrm{pre}, \delta, \mathrm{post}, \mathrm{scope})$ together with the before/after workflow references. The ***effect certificate*** records diagnostic gain and validation outcomes, including guard and cost effects. Given the current failures, the reader filters incompatible records and ranks the remaining matches by relevance, gain, and regression:

$$\begin{aligned} \mathcal{C}_t &= \mathrm{Filter}_{\mathrm{attr}}\big(\mathrm{Filter}_{\mathrm{prov}}(M_t; s_t); \phi_t\big), \\ z_t &= \mathrm{Sort}\big(\mathcal{C}_t; \mathrm{sim}(\phi_t, \phi_i), r_i, -g_i\big). \end{aligned} \tag{7}$$

If $\mathcal{C}_t$ is empty, the editor receives an explicit empty-memory marker and proposes from current evidence alone. The writer is transactional: a record becomes readable iff its corresponding W' is committed, so rejected or incomplete effects cannot influence later rounds. This differs from merely exposing proposal history because later edits receive explicit applicability and effect boundaries. The Flat Accepted History control changes only the representation of accepted records while holding the remaining optimization procedure fixed. It serializes these records as chronological natural language rather than typed transitions. Comparing it with Full therefore isolates the typed interface rather than the availability of successful history.

4.4 Cascaded Validation Gating and Memory Writing

The update rule evaluates $W' = p_t(W_t)$ in increasing order of cost: static legality, path-level smoke execution, diagnostic gain, then guard regression. Let d_t be the number of current failures corrected and q_t the number of current correct guard cases regressed. The operational thresholds are

$$\begin{aligned} \Delta^t_{\min} &= \max\big(1, \lceil 0.03 |D^t_{\mathrm{diag}}| \rceil\big), \\ q_t &\le \min\big\{ \lceil 0.2 |D^t_{\mathrm{guard}}| \rceil, \lfloor 0.5 d_t \rfloor \big\}. \end{aligned} \tag{8}$$

A proposal is accepted iff static and runtime checks pass, $d_t \ge \Delta^t_{\min}$, and the guard bound holds. Acceptance jointly sets $W_{t+1} = W'$ and appends $(\phi_t, p_t, r_t, c_t, g_t)$ to M_t. On rejection, (W_t, M_t) is preserved, and the proposal is retained only for within-round deduplication. Appendix A shows that including all eligible current successes in the guard set makes every accepted update increase the total number of correct training cases by $d_t - q_t \ge 1$. At deployment, only W_T executes, with optimizer memory and the editor absent. Algorithm 1 summarizes the complete procedure. Full gate and cost details appear in Appendices B.5 and 5.6.

Table 1: Main results of three target models on five agent benchmarks. All methods use identical data splits, tool interfaces, and optimization budgets, and report test-set accuracy (ACC, %). Avg is the arithmetic mean across benchmarks. **Bold** and underline mark the best and second-best results.

Models	Methods	OfficeQA	TableEval	SearchQA	ALFWorld	LiveMath	Avg
GPT5.4-Pro	ReAct	48.93	87.12	75.64	75.83	35.91	64.69
	GEPA	61.37	91.83	82.71	74.18	41.92	70.40
	Trace2Skill	52.01	89.71	81.37	72.36	39.53	67.00
	SkillOpt	63.52	92.21	84.08	90.48	43.42	74.74
	AFlow	50.60	88.27	79.05	75.00	38.40	66.26
	Meta-Harness	68.50	91.90	86.60	92.60	44.00	76.72
	ConSeqHarness	**73.33**	**95.55**	**89.42**	**94.86**	**47.61**	**80.15**
Seed2.0-Pro	ReAct	65.12	85.64	89.93	74.29	29.51	68.90
	GEPA	73.33	91.00	92.24	82.23	34.71	74.70
	Trace2Skill	67.04	89.63	91.66	81.23	36.20	73.15
	SkillOpt	75.66	92.22	92.92	89.63	45.34	79.15
	AFlow	66.08	87.63	90.80	77.76	32.11	70.88
	Meta-Harness	77.80	91.80	93.50	92.00	44.80	79.98
	ConSeqHarness	**79.62**	**94.08**	**95.14**	**94.71**	**50.00**	**82.71**
Qwen3.6-Max	ReAct	63.73	85.71	85.69	72.71	22.95	66.16
	GEPA	69.91	91.05	88.64	79.89	28.23	71.54
	Trace2Skill	63.53	89.68	87.87	78.83	28.95	69.77
	SkillOpt	72.24	92.26	89.73	88.43	40.39	76.61
	AFlow	65.22	87.69	86.78	75.77	25.59	68.21
	Meta-Harness	74.80	91.70	90.70	89.60	39.80	77.32
	ConSeqHarness	**78.43**	**95.72**	**93.26**	**91.75**	**43.47**	**80.53**

5 EXPERIMENTS

5.1 EXPERIMENTAL SETUP

We evaluate ConSeqHarness on five public benchmarks: OfficeQA (Opsahl-Ong et al., 2026) for document reasoning, TableEval (Zhu et al., 2025a) for table reasoning, SearchQA (Dunn et al., 2017) for retrieval-based question answering, LiveMathematicianBench (He et al., 2026) (LiveMath) for research-level mathematical reasoning, and ALFWorld (Shridhar et al., 2020) for textual interactive environments. Benchmark details are provided in Appendix C.1.

Our target models are GPT5.4-Pro (OpenAI, 2026a), Seed2.0-Pro (ByteDance Seed, 2026), and Qwen3.6-Max (Qwen Team, 2026). All optimizers use GPT-5.5 (OpenAI, 2026b) as the editor. Optimizable methods share D_{train}, tools, the output parser, and token cap, with a budget of $T = 18$ charged candidate proposals or repairs. The primary comparison uses seed 40. We additionally compare against the strongest harness baseline across three seeds. D_{test} is excluded from diagnosis, retrieval, validation, checkpoint selection, and stopping, and is evaluated only after W_T is frozen. We report task-evaluator accuracy and retain complete optimization artifacts. Appendix C.2 details data splitting and attempt accounting, while Appendix D provides prompts and operational protocols.

5.2 BASELINE METHODS

We compare against fixed ReAct (Yao et al., 2022), the prompt optimizer GEPA (Agrawal et al., 2026), and the skill methods Trace2Skill (Ni et al., 2026) and SkillOpt (Yang et al., 2026). We include the structural optimizers AFlow (Zhang et al., 2025a) and Meta-Harness (Lee et al., 2026). Optimizable baselines share the task specification, target model, evaluator, and charged attempt budget. Non-structural methods keep graph topology and tool interfaces fixed, whereas AFlow and Meta-Harness may modify execution structure. Appendix C.4 details scopes and interface adaptations.

5.3 EXPERIMENTAL RESULTS

Table 1 reports the primary-seed results. ConSeqHarness ranks first in all 15 model–benchmark settings, improving macro-average ACC by 2.73–3.43 points over Meta-Harness. Against the strongest

Table 2: Multi-seed comparison on Seed2.0-Pro. Values are ACC across three data-sampling seeds under the same optimization protocol. Avg is the arithmetic mean across the five benchmarks.

Method	Seed	OfficeQA	TableEval	SearchQA	ALFWorld	LiveMath	Avg
Meta-Harness	40	77.80	91.80	93.50	92.00	44.80	79.98
	80	75.64	92.91	93.21	91.06	45.08	79.58
	2027	76.22	91.37	93.74	91.58	47.36	80.05
ConSeqHarness	40	79.62	94.08	95.14	94.71	50.00	82.71
	80	77.49	92.68	94.90	93.57	49.34	81.60
	2027	78.14	92.05	95.38	94.14	51.24	82.19

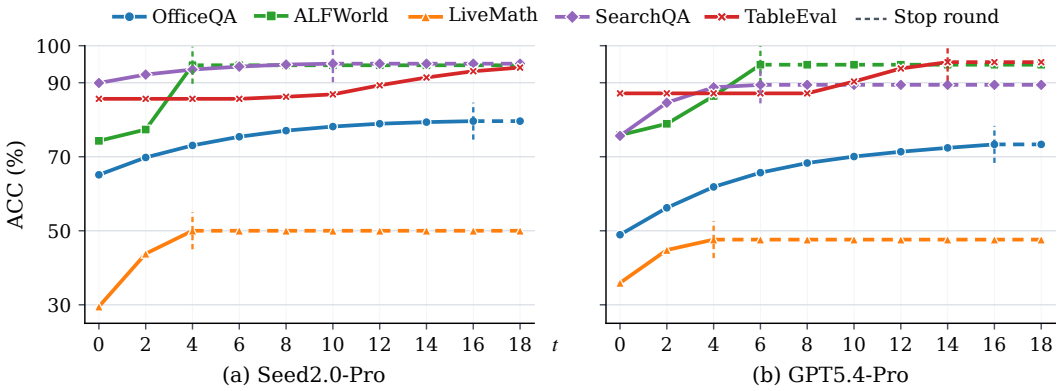


Figure 2: Training-set ACC trends of ConSeqHarness across optimization. The x-axis denotes the charged candidate-attempt index t, and the y-axis denotes ACC on the same fixed D_{train} at every checkpoint. The two subfigures correspond to Seed2.0-Pro and GPT5.4-Pro, respectively.

competing result in each setting, its margin ranges from 1.64 points on Seed2.0-Pro/SearchQA to 4.83 points on GPT5.4-Pro/OfficeQA. The advantage thus spans all benchmark families rather than depending on one target model or task type. Meta-Harness is the strongest baseline in 10 settings, with SkillOpt leading the remaining five, providing complementary structural and non-structural comparators. Example final harnesses are in Appendix E.

5.4 Sampling Robustness

Table 2 reports the paired comparison across three data manifests. The maximum range among ConSeqHarness results is 2.13 points, and its advantage over Meta-Harness is positive for every seed and every reported macro average. The mean paired gain is 2.30 points, showing that the primary result is not driven by one favorable data manifest.

5.5 Long-Horizon Analysis

5.5.1 Round-wise Gain Trends

Figure 2 evaluates every checkpoint on the same fixed D_{train}. All curves are non-decreasing, as implied by the acceptance rule, but exhibit different optimization dynamics. OfficeQA and SearchQA accumulate gradual gains, whereas ALFWorld and LiveMath solve their training cases early. TableEval improves only after several upstream routing and representation edits. These curves verify that accepted edits accumulate rather than oscillate under the training-side gate.

5.5.2 Persistence of Accepted Transitions

For a transition accepted at t_i, structural retention records whether its target edit remains in W_{18}, whereas persistent effectiveness records whether cases newly corrected at t_i remain correct at every later checkpoint. Figure 3 shows that accepted transitions usually remain both structurally present

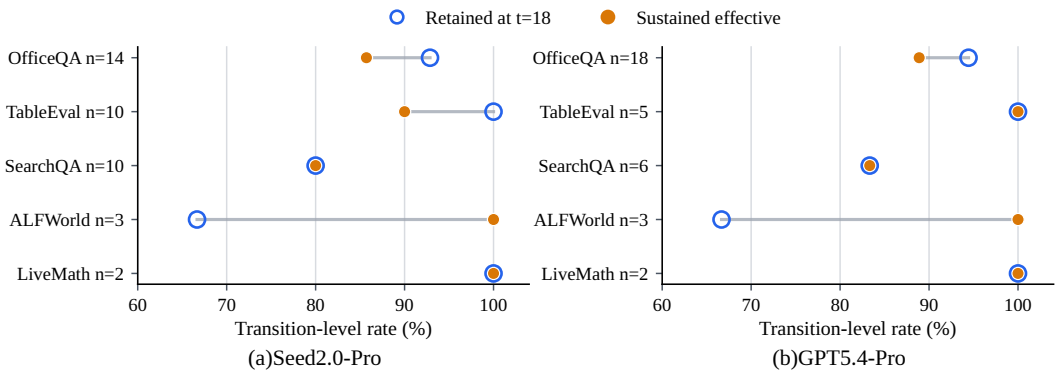


Figure 3: Training-side audit of accepted transitions. Blue denotes structural retention in W_{18}, orange denotes persistent effectiveness on newly corrected cases, and n is the number of accepted transitions. The maximum late regression rate is 0.70%.

Table 3: Retained edit scope and deployment-path cost of W_T over W_0 under Seed2.0-Pro. L1/L2/L3 denote retained intra-node updates, local-branching edits, and terminal-reflection edits, respectively. LLM nodes, token cap, and non-LLM ops are measured on the longest path per invocation.

Benchmark	L1	L2	L3	LLM nodes	Token cap	Non-LLM ops
OfficeQA	1	0	0	$1 \rightarrow 1$	$2048 \rightarrow 2048$	$1 \rightarrow 1$
TableEval	0	11	0	$1 \rightarrow 1$	$4096 \rightarrow 4096$	$1 \rightarrow 23$
SearchQA	2	0	0	$1 \rightarrow 1$	$512 \rightarrow 512$	$2 \rightarrow 2$
ALFWorld	2	0	0	$1 \rightarrow 1$	$2048 \rightarrow 2048$	$2 \rightarrow 2$
LiveMath	1	0	0	$1 \rightarrow 1$	$2048 \rightarrow 2048$	$1 \rightarrow 1$

and behaviorally effective. ALFWorld is the main structural exception. Later edits sometimes replace earlier graph elements, yet persistent effectiveness remains 100%, indicating behavioral consolidation rather than loss of the correction. Across all reported combinations of models and benchmarks, late regression affects at most 0.70% of pairs between newly corrected cases and their associated transitions. Appendix C.5 complements this retention audit with memory-coverage statistics. The matched-history, no-memory, random-retrieval, and write-policy ablations below provide the controlled comparisons.

5.6 Deployment Cost and Scope of Retained Edits

This experiment examines whether the gains of ConSeqHarness come from a larger test-time model budget or from the retained harness edits. We compare the longest executable-path cost of the initial harness W_0 and the final harness W_T on Seed2.0-Pro, and report the scope of edits retained in the final graph relative to W_0. LLM nodes denote model-invocation nodes along the longest executable path. Memory retrieval, candidate generation, and validation are performed only during offline optimization and are therefore excluded from the deployment path. Following the hierarchy in Appendix B.4, retained edits are categorized as L1 intra-node updates, L2 local-branching edits, and L3 terminal-reflection edits.

Table 3 shows unchanged longest-path LLM-node counts and output-token caps. SearchQA and ALFWorld nevertheless retain code/tool mutations, and TableEval retains 11 local branches, increasing its longest-path deterministic operations from 1 to 23. Thus the gains arise from retained execution structure rather than additional LLM calls or a larger generation cap.

5.7 Ablation Experiment

The ablation study evaluates five design choices under the same benchmark set and charged candidate-attempt budget. For every accepted update, Flat Accepted History stores the same failure summary, edit, validation outcome, and guard effect as transition memory, but serializes the record as chrono-

Table 4: Macro-average ablation results. Values are five-benchmark averages. Flat Accepted History serializes the same accepted evidence as chronological natural-language history. GraphEdit+Gate w/o TM keeps graph editing and validation gates but removes transition memory. Δ denotes ACC drop relative to the full method. Eff., Accpt., Reg., and Fail denote effective transition rate, candidate acceptance rate, non-target regression rate, and runtime failure rate.

Target model	Variant	ACC ↑	Δ ↓	Eff. ↑	Accpt. ↑	Reg. ↓	Fail ↓
GPT5.4-Pro	Full	80.15	0.00	62.49	66.42	0.27	1.09
	Flat Accepted History	78.21	1.94	60.88	57.63	1.48	2.21
	GraphEdit+Gate w/o TM	75.41	4.74	59.67	45.37	2.93	3.59
	Random Retrieval	73.47	6.68	54.75	41.08	3.78	4.45
	w/o Verified Write	71.17	8.98	49.83	37.00	7.06	6.60
	Unconstrained Edit	69.86	10.29	45.88	32.37	10.69	11.12
Qwen3.6-Max	Full	80.53	0.00	80.70	66.47	1.24	1.11
	Flat Accepted History	78.86	1.67	72.57	58.94	1.96	2.08
	GraphEdit+Gate w/o TM	76.49	4.04	59.92	45.25	2.97	3.61
	Random Retrieval	74.37	6.16	55.17	41.00	3.80	4.50
	w/o Verified Write	72.36	8.17	50.25	36.92	7.07	6.66
	Unconstrained Edit	71.38	9.15	46.00	32.17	10.66	11.22

logical natural-language history without typed fields. It uses the same GPT-5.5 editor, compatibility filtering, retrieval budget, typed edit operators, repair protocol, and validation gates as the full method. GraphEdit+Gate w/o TM removes transition-memory conditioning while keeping these components unchanged. Random Retrieval preserves the memory pool but shuffles retrieved entries, w/o Verified Write stores candidate attempts without acceptance-gated filtering, and Unconstrained Edit relaxes the typed edit scope. Metric denominators and exact variant definitions are provided in Appendix C.3.

Table 4 separates two effects. Flat Accepted History retains the same accepted evidence, editor, compatibility filter, retrieval budget, operators, and gate but removes typed fields. Its 1.67–1.94 point drop is therefore the most direct estimate of structured representation under this implementation. Removing accepted history entirely causes a larger 4.04–4.74 point drop. Random retrieval shows that irrelevant memory can be worse than no memory, while unverified writing and unconstrained editing sharply increase regression and runtime failures. Thus most benefit comes from retaining relevant accepted evidence, with an additional, smaller gain from representing it as typed transitions.

The reliability metrics follow the same ordering. Full accepts $66.42\%/66.47\%$ of edits for GPT5.4-Pro/Qwen3.6-Max, while keeping regression at $0.27\%/1.24\%$ and runtime failure near 1.1%. Random retrieval reduces acceptance to about 41% and raises both error rates. Without verified writing, regression exceeds 7% and runtime failure exceeds 6.6%. Under unconstrained editing, they rise beyond 10.6% and 11.1%, respectively. The accuracy gains therefore coincide with safer optimization trajectories, rather than merely more aggressive search. Relative to Flat Accepted History, typed transitions raise the effective-transition rate by 1.61 and 8.13 points across the two target models, showing that representation affects edit utility as well as final accuracy.

6 Conclusion

We turn the external execution organization around a frozen model from a one-shot workflow into a continuously improvable structural state. ConSeqHarness retains only local updates that have been executed and validated, and uses these validated experiences to condition later edits. In this way, cross-round feedback becomes more than trajectory logs or textual experience: it forms an optimization state that constrains the edit space, checks side effects, and preserves accumulated gains. Experiments show that this design brings consistent improvements across diverse agent tasks and target models, while improving candidate acceptance, non-target behavior control, and runtime reliability. The current scope focuses on tasks with explicit feedback and structurally executable graphs. Extending this stateful optimization principle to open-ended online environments, multi-agent execution, and long-term deployment remains an important direction for future work.

AI USE STATEMENT

We used generative AI both as part of the proposed method and in the experimental pipeline. Specifically, GPT-5.5 served as the fixed editor for proposing harness modifications, while the reported target models were used in the experiments. All AI-generated modifications were evaluated through the validation procedures described in Sections 4 and Section 5, and the held-out test sets were never used for optimization or model selection. Outside the experimental pipeline, we used generative AI only for textual polishing. We developed and verified the research methodology, experimental protocol, scientific claims, and interpretation of results, reviewed all AI-assisted content, and take responsibility for the final manuscript and results.

REPRODUCIBILITY STATEMENT

We provide detailed implementation and evaluation specifications to facilitate reproducibility. Sections A formalize the optimizer state, memory updates, and operational guarantees; Sections B describe the executable harness semantics, initialization, transition retrieval, constrained edit operators, and validation gates; Sections C specify the benchmarks, data splits, budget accounting, ablation protocols, baseline implementations, and transition diagnostics; and Section D provides the prompts and operational protocols used in our experiments. All optimizable methods use the same data splits, target models, tools, evaluator, output parser, token cap, and $T = 18$ charged candidate-attempt budget, with the held-out test set used only for final evaluation. The anonymized code is available at `https://anonymous.4open.science/r/ConSeqHarness-579E`.

REFERENCES

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, et al. Gepa: Reflective prompt evolution can outperform reinforcement learning. In *International Conference on Learning Representations*, volume 2026, pp. 8479–8565, 2026.

ByteDance Seed. Seed2.0 model card: Towards intelligence frontier for real-world complexity, February 2026.

Matthew Dunn, Levent Sagun, Mike Higgins, V Ugur Guney, Volkan Cirik, and Kyunghyun Cho. Searchqa: A new q&a dataset augmented with context from a search engine. *arXiv preprint arXiv:1704.05179*, 2017.

Hongcheng Gao, Yue Liu, Yufei He, Longxu Dou, Chao Du, Zhijie Deng, Bryan Hooi, Min Lin, and Tianyu Pang. Flowreasoner: Reinforcing query-level meta-agents. *arXiv preprint arXiv:2504.15257*, 2025.

Linyang He, Qiyao Yu, Hanze Dong, Baohao Liao, Xinxing Xu, Micah Goldblum, Jiang Bian, and Nima Mesgarani. Livemathematicianbench: A live benchmark for mathematician-level reasoning with proof sketches. *arXiv preprint arXiv:2604.01754*, 2026.

Seth Karten, Joel Zhang, Tersoo Upaa Jr, Ruirong Feng, Wenzhe Li, Chengshuai Shi, Chi Jin, and Kiran Vodrahalli. Continual harness: Online adaptation for self-improving foundation agents. *arXiv preprint arXiv:2605.09998*, 2026.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses. *arXiv preprint arXiv:2603.28052*, 2026.

Xiaochuan Li, Ryan Ming, Pranav Setlur, Abhijay Paladugu, Andy Tang, Hao Kang, Shuai Shao, Rong Jin, and Chenyan Xiong. Benchmark test-time scaling of general llm agents. *arXiv preprint arXiv:2602.18998*, 2026.

Zelong Li, Shuyuan Xu, Kai Mei, Wenyue Hua, Balaji Rama, Om Raheja, Hao Wang, He Zhu, and Yongfeng Zhang. Autoflow: Automated workflow generation for large language model agents. *arXiv preprint arXiv:2407.12821*, 2024.

Huawei Lin, Peng Li, Jie Song, Fuxin Jiang, and Tieying Zhang. Muse-autoskill: Self-evolving agents via skill creation, memory, management, and evaluation. *arXiv preprint arXiv:2605.27366*, 2026.

Lin Long, Yichen He, Wentao Ye, Yiyuan Pan, Yuan Lin, Hang Li, Junbo Zhao, and Wei Li. Seeing, listening, remembering, and reasoning: A multimodal agent with long-term memory. In *International Conference on Learning Representations*, volume 2026, pp. 146197–146246, 2026.

Jingwei Ni, Yihao Liu, Xinpeng Liu, Yutao Sun, Mengyu Zhou, Pengyu Cheng, Dexin Wang, Erchao Zhao, Xiaoxi Jiang, and Guanjun Jiang. Trace2skill: Distill trajectory-local lessons into transferable agent skills. *arXiv preprint arXiv:2603.25158*, 2026.

Boye Niu, Yiliao Song, Kai Lian, Yifan Shen, Yu Yao, Kun Zhang, and Tongliang Liu. Flow: Modularized agentic workflow automation. In *International Conference on Learning Representations*, volume 2025, pp. 74949–74977, 2025.

OpenAI. Introducing GPT-5.4, March 2026a. URL https://openai.com/index/introducing-gpt-5-4/.

OpenAI. Introducing GPT-5.5, April 2026b. URL https://openai.com/index/introducing-gpt-5-5/.

Krista Opsahl-Ong, Michael J Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. Optimizing instructions and demonstrations for multi-stage language model programs. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 9340–9366, 2024.

Krista Opsahl-Ong, Arnav Singhvi, Jasmine Collins, Ivan Zhou, Cindy Wang, Ashutosh Baheti, Owen Oertell, Jacob Portes, Sam Havens, Erich Elsen, et al. Officeqa pro: An enterprise benchmark for end-to-end grounded reasoning. *arXiv preprint arXiv:2603.08655*, 2026.

Linyue Pan, Lexiao Zou, Shuo Guo, Jingchen Ni, and Hai-Tao Zheng. Natural-language agent harnesses. *arXiv preprint arXiv:2603.25723*, 2026.

Qwen Team. Qwen3.6-Max-Preview: Smarter, sharper, still evolving, April 2026. URL https://qwen.ai/blog?id=qwen3.6-max-preview.

Junhong Shen, Hao Bai, Lunjun Zhang, Yifei Zhou, Amrith Setlur, Shengbang Tong, Diego Caples, Nan Jiang, Tong Zhang, Ameet Talwalkar, et al. Thinking vs. doing: Agents that reason by scaling test-time interaction. *arXiv preprint arXiv:2506.07976*, 2025.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. Alfworld: Aligning text and embodied environments for interactive learning. *arXiv preprint arXiv:2010.03768*, 2020.

Jun Wang. Memento-ii: Learning by stateful reflective memory. *arXiv preprint arXiv:2512.22716*, 2025.

Yanbo Wang, Zixiang Xu, Yue Huang, Xiangqi Wang, Zirui Song, Lang Gao, Chenxi Wang, Robert Tang, Yue Zhao, Arman Cohan, et al. Dyflow: Dynamic workflow framework for agentic reasoning. *Advances in Neural Information Processing Systems*, 38:174148–174181, 2026a.

Yinjie Wang, Ling Yang, Guohao Li, Mengdi Wang, and Bryon Aragam. Scoreflow: Mastering llm agent workflows via score-based preference optimization. *arXiv preprint arXiv:2502.04306*, 2025.

Yinjie Wang, Xuyang Chen, Xiaolong Jin, Mengdi Wang, and Ling Yang. Openclaw-rl: Train any agent simply by talking. *arXiv preprint arXiv:2603.10165*, 2026b.

Shengxiang Xu, Jiayi Zhang, Shimin Di, Yuyu Luo, Liang Yao, Hanmo Liu, Jia Zhu, Fan Liu, and Min-Ling Zhang. Robustflow: Towards robust agentic workflow generation. *arXiv preprint arXiv:2509.21834*, 2025.

Yifan Yang, Ziyang Gong, Weiquan Huang, Qihao Yang, Ziwei Zhou, Zisu Huang, Yan Li, Xuemei Gao, Qi Dai, Bei Liu, et al. Skillopt: Executive strategy for self-evolving agent skills. *arXiv preprint arXiv:2605.23904*, 2026.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Jiabo Ye, Xi Zhang, Haiyang Xu, Haowei Liu, Junyang Wang, Zhaoqing Zhu, Ziwei Zheng, Feiyu Gao, Junjie Cao, Zhengxi Lu, et al. Mobile-agent-v3: Fundamental agents for gui automation. *arXiv preprint arXiv:2508.15144*, 2025.

Hongli Yu, Tinghong Chen, Jiangtao Feng, Jiangjie Chen, Weinan Dai, Qiying Yu, Ya-Qin Zhang, Wei-Ying Ma, Jingjing Liu, Mingxuan Wang, et al. Memagent: Reshaping long-context llm with multi-conv rl-based memory agent. In *International Conference on Learning Representations*, volume 2026, pp. 39458–39486, 2026.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. Textgrad: Automatic" differentiation" via text. *arXiv preprint arXiv:2406.07496*, 2024.

Yunpeng Zhai, Shuchang Tao, Cheng Chen, Anni Zou, Ziqian Chen, Qingxu Fu, Shinji Mai, Li Yu, Jiaji Deng, Zouying Cao, et al. Agentevolver: Towards efficient self-evolving agent system. *arXiv preprint arXiv:2511.10395*, 2025.

Hangfan Zhang, Shao Zhang, Kangcong Li, Chen Zhang, Yang Chen, Yiqun Zhang, Lei Bai, and Shuyue Hu. Self-harness: Harnesses that improve themselves. *arXiv preprint arXiv:2606.09498*, 2026a.

Hanrong Zhang, Shicheng Fan, Henry Peng Zou, Yankai Chen, Zhenting Wang, Jiayu Zhou, Chengze Li, Wei-Chieh Huang, Yifei Yao, Kening Zheng, et al. Coevoskills: Self-evolving agent skills via co-evolutionary verification. *arXiv preprint arXiv:2604.01687*, 2026b.

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

Jusheng Zhang, Kaitong Cai, Qinglin Zeng, Ningyuan Liu, Stephen Fan, Ziliang Chen, and Keze Wang. Failure-driven workflow refinement. *arXiv preprint arXiv:2510.10035*, 2025b.

Junnan Zhu, Jingyi Wang, Bohan Yu, Xiaoyu Wu, Junbo Li, Lei Wang, and Nan Xu. Tableeval: A real-world benchmark for complex, multilingual, and multi-structured table question answering. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 7137–7157, 2025a.

King Zhu, Hanhao Li, Siwei Wu, Tianshun Xing, Dehua Ma, Xiangru Tang, Minghao Liu, Jian Yang, Jiaheng Liu, Yuchen Eleanor Jiang, et al. Scaling test-time compute for llm agents. *arXiv preprint arXiv:2506.12928*, 2025b.

A Optimizer State and Operational Guarantees **13**

A.1 Setup 13

A.2 Memory-Augmented State Sufficiency and Closure 14

A.3 Feasibility and Certified Memory Writes 15

A.4 Discussion 15

B Implementation Semantics of the Agent Execution Harness **15**

B.1 Executable Harness Nodes and Structural Constraints 16

B.2 Initial Harness Synthesis Interface 17

B.3 Provenance-Aware Memory Retrieval and Serialization 17

B.4 Constrained Operators and Edit Hierarchy 18

B.5 Risk-Progressive Validation Gating 18

C Experimental Detail **19**

C.1 Benchmarks 19

C.2 Budget and Cost Alignment 19

C.3 Operational Definitions and Ablation Protocol 20

C.4 Baseline Methods 20

C.5 Transition Diagnostics 22

D Prompt and Skills **22**

E Examples of Optimized Harnesses **30**

A Optimizer State and Operational Guarantees

This appendix formalizes the algorithmic properties induced by the memory-conditioned read–write update in ConSeqHarness. The analysis explains why evaluated transition memory must be included in the optimization state and shows that accepted updates preserve harness feasibility and carry explicit empirical certificates. These properties concern the sequential optimizer itself, focusing on state evolution, feasibility preservation, and validation-certified updates.

A.1 Setup

Let

$$H_t = (\bar{s}_0, p_0, \ldots, \bar{s}_{t-1}, p_{t-1}) \tag{9}$$

denote the optimization history before round t. We define

$$s_t = (W_t, \tau_t, e_t, D^t_{\text{diag}}, D^t_{\text{guard}}, \kappa_t), \quad \bar{s}_t = (s_t, M_t), \tag{10}$$

where W_t is the current harness, τ_t and e_t are its training-side trajectories and evidence, and M_t stores evaluated historical transitions. The operational context $\kappa_t = (t, b_t, D^t_{\text{smoke}}, u_t)$ records the round index, remaining charged candidate-attempt budget, current smoke set, and short-term rejected-candidate deduplication state. Fixed editor, evaluator, and decoding configurations are treated as part of the transition environment. Under the fixed read–write algorithm,

$$\phi_t = \Phi(e_t), \quad z_t \sim \rho(\cdot \mid s_t, M_t), \quad p_t \sim K_\theta(\cdot \mid s_t, z_t). \tag{11}$$

Algorithm 1 ConSeqHarness optimization with atomic transition writes

Require: Initial feasible harness W_0, training set D_{train}, charged-attempt budget T
$(W, M, b, u) \leftarrow (W_0, \varnothing, T, \varnothing)$ ▷ accepted state, budget, rejected-edit cache
while $b > 0$ **do**
  $(\tau, e) \leftarrow \text{Execute}(W, D_{\text{train}})$
  Construct D_{diag} from all current failures, D_{guard} from all current successes, and D_{smoke}
  if $D_{\text{diag}} = \varnothing$ **then**
    break
  end if
  $s \leftarrow (W, \tau, e, D_{\text{diag}}, D_{\text{guard}}, (T - b, b, D_{\text{smoke}}, u))$; $\bar{s} \leftarrow (s, M)$
  if $\mathcal{A}(\bar{s}) = \varnothing$ **then**
    break
  end if
  $\phi \leftarrow \Phi(e)$; retrieve $z \sim \rho(\cdot \mid s, M)$, using an empty marker if no record is compatible
  Propose scoped edit $p \sim K_\theta(\cdot \mid s, z)$; $b \leftarrow b - 1$; $W' \leftarrow p(W)$
  if $\neg\,\text{StaticPass}(W') \wedge b > 0$ **then**
    $p \leftarrow \text{Repair}(p)$; $b \leftarrow b - 1$; $W' \leftarrow p(W)$
  end if
  $\Delta \leftarrow \max\{1, \lceil .03|D_{\text{diag}}| \rceil\}$; $\eta(d) \leftarrow \min\{\lceil .2|D_{\text{guard}}| \rceil, \lfloor .5d \rfloor\}$
  $(a, c) \leftarrow \text{CascadedGate}(W, W'; D_{\text{smoke}}, D_{\text{diag}}, D_{\text{guard}}, \Delta, \eta)$ ▷ static → runtime → gain → guard
  if $a = 1$ **then**
    Form s'; set $r \leftarrow R(s, p, s')$ and $g \leftarrow \text{GuardEffect}(c)$
    $(W, M, u) \leftarrow \big(W', \text{Append}(M, (\phi, p, r, c, g)), \varnothing\big)$ ▷ atomic commit and write
  else
    $u \leftarrow u \cup \{\text{sig}(p)\}$ ▷ reject: keep (W, M) unchanged
  end if
end while
return current accepted harness W

Applying p_t produces a candidate harness, whose execution and validation yield reward r_t, check record c_t, and guard effect g_t. The memory update is

$$M_{t+1} = U(M_t, \phi_t, p_t, r_t, c_t, g_t). \tag{12}$$

All randomness from the editor, tools, and environment is included in the corresponding conditional distributions.

A.2 Memory-Augmented State Sufficiency and Closure

Proposition 1 (Memory-Augmented State Sufficiency and Closure)**.** *Assume that the retrieval policy can return different memory fragments for two histories with the same s_t but different M_t, and that the editor is not invariant to all such fragments. Then s_t alone is insufficient to characterize the next-edit distribution. Under a fixed execution, validation, split-refresh, and memory-update interface, the augmented state $\bar{s}_t = (s_t, M_t)$ is closed with respect to the next optimization transition:*

$$\Pr(\bar{s}_{t+1} \mid \bar{s}_{0:t}, p_{0:t}) = \Pr(\bar{s}_{t+1} \mid \bar{s}_t, p_t). \tag{13}$$

Proof. Consider histories H_t and H'_t that induce the same current harness state s_t but different memories $M_t \neq M'_t$. Their induced edit distributions are

$$\pi_\theta(p \mid s_t, M_t) = \sum_z \rho(z \mid s_t, M_t) K_\theta(p \mid s_t, z). \tag{14}$$

By assumption, the memories induce different retrieval distributions and the editor responds to at least one differing fragment. Hence the two mixtures differ for some edit p, so s_t alone is not sufficient.

Given $\bar{s}_t$ and p_t, the candidate is determined by $\text{Apply}(W_t, p_t)$. The fixed execution environment, validation cascade, and manifest-backed split-refresh policy then determine the conditional distribution of the next trajectories, evidence, and accepted harness. The memory update depends only on M_t, the current failure summary, the proposed edit, and the resulting validation record. Therefore, once $(\bar{s}_t, p_t)$ is given, earlier history does not provide additional information required by the algorithm to generate $\bar{s}_{t+1}$, which establishes the stated closure. □

A.3 Feasibility and Certified Memory Writes

Proposition 2 (Feasibility and Certified Memory Writes). *Assume that W_0 satisfies the structural constraint set C, and that $\mathrm{StaticPass}(W) = 1$ is sound with respect to C. If ConSeqHarness commits and writes a candidate only after it passes the validation gate, then every accepted harness W_t satisfies C. Moreover, every transition m_t written to memory satisfies*

$$\mathrm{diag_gain}_t \geq \Delta^t_{\min} = \max\big(1, \lceil 0.03|D^t_{\mathrm{diag}}| \rceil\big)\,, \tag{15}$$

$$\mathrm{Regress}_t \leq \min\big\{ \lceil 0.2|D^t_{\mathrm{guard}}| \rceil\,, \lfloor 0.5\,\mathrm{diag_gain}_t \rfloor \big\}\,. \tag{16}$$

Thus, each retrievable memory entry is paired with a write-time feasibility and behavioral-validation certificate.

Proof. The claim follows by induction over accepted updates. The base harness W_0 satisfies C. A rejected candidate leaves the accepted harness unchanged, whereas an accepted candidate must satisfy $\mathrm{StaticPass}(W') = 1$, implying $W' \in C$ by soundness. Therefore every accepted W_t remains structurally feasible. A transition enters long-term memory only when the same candidate also meets the diagnostic-gain floor and both guard-regression constraints. Rejected candidates are excluded from long-term memory. Consequently, every retrievable transition carries the displayed write-time certificate. □

Proposition 3 (Monotonic Training-Side Progress). *Suppose D^t_{diag} contains every example currently incorrect under W_t, and D^t_{guard} contains every example currently correct under W_t. Let C_t be the number of correct examples in the fixed training set. Every accepted update satisfies*

$$C_{t+1} - C_t = \mathrm{diag_gain}_t - \mathrm{Regress}_t \geq \lceil \tfrac{1}{2}\mathrm{diag_gain}_t \rceil \geq 1. \tag{17}$$

Consequently, the accepted sequence is strictly improving on the fixed training set and contains at most $|D_{\mathrm{train}}| - C_0$ accepted transitions.

Proof. The diagnostic and guard sets partition current errors and successes, so the change in the number correct is exactly the number of corrected errors minus the number of regressed successes. The acceptance rule gives $\mathrm{Regress}_t \leq \lfloor 0.5\,\mathrm{diag_gain}_t \rfloor$, while the diagnostic-gain floor is at least one. The displayed inequality follows. Since C_t is an integer bounded above by $|D_{\mathrm{train}}|$, the number of accepted updates is at most $|D_{\mathrm{train}}| - C_0$. □

Gate errors. The certificate is empirical and depends on the validation gate. Let E_t be the event that the gate falsely accepts a write-time harmful transition at round t, and let $\alpha_t = \Pr(E_t)$. Because a harmful transition can enter memory only through such a false acceptance,

$$\Pr[M_T \text{ contains a write-time harmful transition}] \leq \sum_{t=0}^{T-1} \alpha_t \tag{18}$$

by the union bound. This observation makes the dependence on validator quality explicit and complements the feasibility analysis above.

A.4 Discussion

These results characterize the invariants of the ConSeqHarness optimization algorithm. The first proposition identifies when evaluated transition memory is a necessary component of the algorithmic state and establishes closure under the fixed update interface. The second shows that the accepted sequence remains within the feasible harness space and that every reusable transition is accompanied by explicit training-side validation evidence. Together, they provide formal grounding for memory-augmented state construction and validation-gated harness updates.

B Implementation Semantics of the Agent Execution Harness

This appendix details the concrete state interface, syntactic constraints, and verification protocols of the executable agent harness framework in ConSeqHarness. These semantics provide a reproducible and isolated representation of workflow optimization independent of task-specific logic.

B.1 Executable Harness Nodes and Structural Constraints

We implement the execution harness as a strongly-typed, directed program graph. Topologically, the graph is a map from unique node identifiers to discrete programmatic node definitions. Every node $v \in V$ explicitly registers a unique name, a specific primitive type, structural configuration parameters, named input references, declared output schemas, and valid downstream successor edges. The execution environment maintains a global routing table indexed by node names, enabling full runtime trace reconstruction including node-local inputs, localized execution paths, unhandled errors, and terminal outputs.

We categorize the functional primitives of the node set V into seven canonical classes based on their execution semantics and roles in the state-transition optimization loop:

- **Entry Nodes:** These nodes ingest raw environment observations or user queries and project them into structured, named field schemas accessible to downstream computational scopes. They serve as the static root of the dataflow graph, establishing the lifetime invariants of input tensors or text fragments.
- **Model Nodes:** These nodes invoke models by injecting upstream data fields into local structural prompting templates, forcing the model to generate a statically parseable JSON or dictionary schema. In the context of optimization, model nodes expose highly mutable handles, such as prompt instructions and few-shot exemplars, which are tuned by the editing policy to resolve cognitive or reasoning failures identified during trace diagnosis.
- **Code Nodes:** These nodes execute deterministic and stateless computation primitives to perform data parsing, regular expression extraction, intermediate field mapping, or strict type conversions. They act as semantic bridges between noisy model outputs and rigid programmatic requirements, ensuring that downstream routing or execution components receive clean inputs that strictly satisfy static type contracts.
- **Routing Nodes:** These nodes compute conditional branching selections over upstream structural inputs to dynamically steer the graph control flow along distinct execution trajectories. They represent the non-linear decision boundaries of the execution harness, and their optimization requires the editing policy to adjust conditional thresholds or introduce exception handlers without breaking global reachability constraints.
- **Conclusion Nodes:** These nodes generate task-level intermediate decisions, reflection tokens, or critical verification milestones during specific execution checkpoints. Unlike final answers, conclusion nodes externalize the hidden states of multi-step reasoning processes into inspectable execution logs, providing explicit tracing markers that allow the framework diagnoser to isolate the exact step where an error propagates.
- **Terminal Nodes:** These nodes map the accumulated intermediate decisions and verification histories from upstream conclusion nodes into the final canonical task output schema required by the environment evaluation metric. They define the structural termination boundary of the harness and enforce strict compliance with target formatting rules, ensuring that different candidate graphs can be rigorously compared within the identical evaluation space.
- **Parallel and Loop Nodes:** These nodes express independent sub-task concurrency or bounded local iterations, which define the structural capabilities of the architecture but remain non-essential for basic graph optimization. They manage complex control patterns such as batch map-reduce tool invocations or progressive self-correction loops, requiring strict parent-child synchronization boundaries to maintain topological stability.

To guarantee that the state space remains checkable during optimization, the graph state must strictly satisfy the following structural invariants, which materialize the abstract constraints C formulated in Section 3: *First*, node parameter schemas must match their declared primitive types exactly. *Second*, input dependencies must explicitly reference existing node names and valid output schemas instead of arbitrary identifier hashes, and all non-system node names must be unique. *Third*, the graph topology must be fully connected and reachable from the designated entry node, restricting cycles except for multi-path access at designated terminal or synchronization junctions. *Fourth*, the conditional branch targets of any routing node must be bijective to its successor edge set and register a default fallback path. These constraints eliminate arbitrary text mutations, framing the optimization process as a structurally bounded and inspectable state-transition sequence.

B.2 Initial Harness Synthesis Interface

When a seed harness W_0 is absent, ConSeqHarness compiles an initial baseline graph from the high-level task specification. The synthesis interface enforces three core architectural properties: single node responsibility, explicit named field data passing, and the externalization of complex logic into deterministic code or routing nodes rather than overloading single prompts.

The automated bootstrapping protocol proceeds through six sequential phases:

1. **Specification Extraction:** Parsed signals from the task description isolate the target objectives, input fields, output schemas, available environment tools, and environmental hard bounds.
2. **Primitive Decomposition:** The parsed objective is broken down into atomic steps comprising extraction, tool calls, algorithmic computation, and terminal formatting.
3. **Scope Filtration:** The entry signature exposes only necessary fields to downstream scopes, filtering out task-irrelevant context to prevent token bloat.
4. **Topological Planning:** The synthesis engine establishes a baseline structure (e.g., sequential pipelines, coarse-to-fine routing) tailored to the target task difficulty.
5. **Node Instantiation:** Every atomic primitive is cast to its corresponding node schema, initializing prompting templates, parsing functions, and output configurations.
6. **Executability Acceptance:** The synthesized graph undergoes structural checking and single-example evaluation runs to verify that $\mathrm{StaticPass}(W_0) = 1$.

This protocol explicitly bars unstable initial states, such as multi-objective prompt crowding or implicit routing within unparsed text blocks. The initial synthesis guarantees state executability, ensuring that the tracking of W_0 provides localizable evidence for subsequent diagnosis and search.

B.3 Provenance-Aware Memory Retrieval and Serialization

Memory retrieval queries strictly verified state-transition logs rather than raw datasets, historical traces, or unstructured instruction text. At each optimization round, the system compresses training failures into a canonical summary ϕ_t containing the failure archetype, targeted node context, trigger boundaries, and stable dependency markers. This structured index serves as the search query, avoiding task-specific data leakage or token context bloat.

The cumulative memory M_t is serialized as an append-only JSONL transaction ledger, where each entry $m_i \in M_t$ records a successfully verified state mutation:

```
m_i = {
  "iteration": 14,
  "phi": {
    "failure_type": "SchemaMismatch",
    "node": "FlightSearch",
    "trigger": "budget_overflow"
  },
  "p": {
    "operator": "OutputConstraintEdit",
    "target": "FlightSearch",
    "delta": "add_pre_filter"
  },
  "r": { "diagnostic_reward": 0.12, "metric_delta": 0.08 },
  "c": { "static_pass": true, "runtime_pass": true },
  "g": { "guard_regression": 0.00, "cost_delta": 0.02 },
  "workflow_before": { ... },
  "workflow_after": { ... }
}
```

The retrieval operator ρ parses M_t via a strict three-tier hierarchical sieve: *The first* tier enforces strict metadata provenance matching denoted by $\mathrm{Filter}_{\mathrm{prov}}$ to eliminate entries with incompatible

workflow configurations, environment schemas, or output evaluation dimensions. *The second* tier restricts the candidates to attribute matches denoted by $\mathrm{Filter}_{\mathrm{attr}}$ by filtering for overlapping failure types and identical target node locations. *The third* tier sorts the remaining pool via the Sort operator to prioritize verified transitions that achieved maximal diagnostic reward r_i and zero guard regression.

To reduce the risk of amplifying incompatible historical transitions through retrieval, the reader reuses only accepted transitions whose provenance is compatible with the setting. A candidate memory must match the current task mode, workflow schema, output contract, evaluation setup, and task-specific configuration. The memory read budget is defined by compatibility filtering: all accepted transitions that pass the provenance and attribute filters are supplied to the editor together with their validation records. If no entry passes these filters, ρ returns an empty-memory state, forcing the policy to generate an edit from the current failure summary. This design bounds memory conditioning to past verified state transitions and isolates the search policy from downstream test variables.

B.4 Constrained Operators and Edit Hierarchy

The admissible action space $\mathcal{A}(\bar{s}_t)$ restricts graph mutations to localized, typed edit operators. Each candidate operator $p \in \mathcal{A}(\bar{s}_t)$ declares explicit execution preconditions, structural postconditions, and a bounded modification scope.

Editor proposal protocol. At each round, the LLM editor receives the current failure summary ϕ_t, the retrieved memory fragment z_t or an explicit empty-memory marker, diagnostic and guard evidence, and the current harness schema. The editor must emit a constrained edit record

$$p_t = (\mathrm{op}, \mathrm{target}, \mathrm{pre}, \delta, \mathrm{post}, \mathrm{scope}), \tag{19}$$

where *op* is one of the operator families below, *target* names the node, field, or edge being changed, *pre* states conditions that must already hold, δ is the minimal semantic update, *post* states the required invariant after applying the edit, and *scope* lists affected paths and non-target boundaries. The structural edit budget is specified by this declared scope: each proposal must localize to one affected path or node boundary, and multiple primitive changes are committed together only when they share the same scope and pass the same validation gate. Invalid graph JSON, missing edit fields, or edits that cannot be localized to a single declared scope are rejected before metric evaluation. A candidate may be repaired once before validation only when its invalidity is a direct schema consequence of the proposed edit. Otherwise, the attempt is counted as rejected and M_t remains unchanged.

Operator Families. The optimization kernel supports six discrete operator types: *First*, output constraint adjustments tune model node schemas or terminal formatting maps to stabilize downstream consumption. *Second*, node semantic modifications fine-tune local prompting instructions or in-context exemplars without shifting the global topology. *Third*, code mutations update data-parsing logic or algorithmic transformations while maintaining type contract safety. *Fourth*, routing boundary edits shift conditional parameters inside routing nodes to match observed failure triggers. *Fifth*, local branching insertions construct minor validation or exception paths adjacent to a failing node. *Sixth*, terminal consistency loops introduce consensus nodes preceding final submission.

To guard against catastrophic semantic collapse, the search space enforces an explicit edit hierarchy, compelling the editor to choose the lowest effective scope capable of resolving the current failure mode. *Level 1* restricts updates entirely within single node configuration boundaries without altering graph variables V and E. *Level 2* permits local sub-graph branching by inserting conditional validation filters adjacent to a faulty node, requiring strict routing completeness and fallback retention. *Level 3* authorizes global consensus or terminal reflection nodes immediately preceding the graph termination to stabilize noisy decision mappings. Structural mutations trigger a cascading update across dependent parent-child links, input references, and routing targets, which must be verified simultaneously by the validation architecture.

B.5 Risk-Progressive Validation Gating

The validation gate evaluates candidate mutations $W' = p_t(W_t)$ through a risk-progressive pipeline, pruning malformed structures prior to costly empirical execution runs. The verification cascade executes the following sequential steps:

- **Static Legality Verification:** The validator checks configuration signatures, data references, graph connectivity, route completeness, and terminal schema constraints, confirming that $\mathrm{StaticPass}(W') = 1$ and that the terminal output domain remains within valid task bounds.
- **Path-Level Environment Smoke Testing:** For structural or routing alterations, the framework runs target executions over the smoke set D_{smoke} to verify that the modified paths execute without runtime faults, ensuring $\mathrm{RuntimePass}(W', D_{\mathrm{smoke}}) = 1$.
- **Diagnostic Improvement Estimation:** The candidate is evaluated on all current failures in D^t_{diag} and must correct at least $\Delta^t_{\min} = \max(1, \lceil 0.03|D^t_{\mathrm{diag}}| \rceil)$ cases.
- **Guard Regression Control:** Let d_t be diagnostic gain and q_t the number of regressed current successes. The gate requires $q_t \leq \min\{\lceil 0.2|D^t_{\mathrm{guard}}| \rceil, \lfloor 0.5 d_t \rfloor\}$.

A candidate transition is committed $W_{t+1} = W'$ if and only if all progressive criteria are satisfied with $\mathbb{I}_{\mathrm{accept}} = 1$, triggering a long-term certificate write into M_{t+1}. Malformed, error-prone, or regressive candidates are immediately aborted, rolling the state back to $W_{t+1} = W_t$. Denied transitions are permanently barred from the long-term memory ledger, serving exclusively for short-term intra-iteration candidate deduplication.

C Experimental Detail

C.1 Benchmarks

We evaluate ConSeqHarness on five benchmarks spanning structured evidence reading, table reasoning, retrieval-based question answering, research-level mathematical reasoning, and interactive environment control, all with verifiable task feedback for harness-level reflection and decision making.

OfficeQA (Opsahl-Ong et al., 2026) evaluates grounded reasoning over real-world office documents, where agents must locate relevant evidence from documents, tables, charts, or text and produce a single verifiable answer.

TableEval (Zhu et al., 2025a) evaluates table-based question answering across diverse languages, structures, and domains, requiring agents to parse table layouts, align field semantics, and perform aggregation, comparison, or semantic reasoning.

SearchQA (Dunn et al., 2017) evaluates retrieval-based question answering, where agents retrieve external evidence, filter relevant information, and synthesize a verifiable answer.

LiveMath (He et al., 2026) follows LiveMathematicianBench and evaluates research-level mathematical reasoning over recent arXiv papers by converting theorems into proof-sketch-conditioned multiple-choice questions. We map each final choice to binary correctness and report ACC.

ALFWorld (Shridhar et al., 2020) evaluates interactive task completion in household environments, where agents select actions from partial observations and receive task-completion feedback.

Each benchmark is split into a training set D_{train} and a test set D_{test}. For benchmarks with an official training split, we construct D_{train} and D_{test} from the training and validation portions, respectively, under a unified sampling protocol. For benchmarks that only provide a test split, we partition the original test set into D_{train} and D_{test}. In ConSeqHarness, D_{train} is further divided into two disjoint subsets, D_{diag} and D_{guard}. D_{smoke} is sampled from D_{diag} and used for basic execution checks of candidate harnesses. The sampling protocol follows the setup of SkillOpt (Yang et al., 2026).

C.2 Budget and Cost Alignment

Split and cost protocol. We follow SkillOpt in constructing benchmark splits from each task's native example pool. When proportional splitting is reliable, two thirds of the examples are used as D_{train} and one third is reserved for held-out reporting. For smaller pools, where proportional subsplits would make diagnostic or guard checks unstable, we use fixed manifest-backed splits instead of further subsampling. D_{test} is kept out of diagnosis, retrieval, and edit generation, and is used only for final reporting. After each accepted state update, the current workflow is rerun on D_{train}: remaining failures define D^t_{diag}, current successes, previously fixed cases, and stable non-target cases define D_{guard}, and one target-path plus one non-target-path example are used for D^t_{smoke} when

path distinctions are available. All selected case IDs are stored in per-round split manifests for reproducibility, and all optimizable methods share the same D_{train}, target model, tool set, output parser, token cap, and $T = 18$ charged candidate-attempt budget. For offline cost accounting, every optimization attempt that proposes, revises, or directly repairs a candidate workflow is charged to this shared candidate-attempt budget, regardless of whether it is accepted, rejected, or regenerated. A schema-valid scored candidate is evaluated through the same cascade over D^t_{smoke}, D^t_{diag}, and D_{guard}. Infrastructure retries that do not change the candidate or sampled validation sets are logged but not counted as new candidate attempts.

Optimizer parameters. Unless otherwise specified by a benchmark manifest, we use deterministic data-sampling seed 40, do not subsample D_{train}, and set both diag_batch_size and guard_sample_size to `all`. Thus each round uses all current training failures for D^t_{diag} and all eligible current successes, previously fixed cases, and stable non-target cases for D_{guard}. Candidate edits are evaluated sequentially, with at most one accepted state update per optimization round. A candidate must satisfy the diagnostic gain floor $\Delta^t_{\min} = \max(1, \lceil 0.03|D^t_{\text{diag}}| \rceil)$ and the two guard constraints $\text{Regress} \le \lceil 0.2|D_{\text{guard}}| \rceil$ and $\text{Regress} \le \lfloor 0.5\,\text{diag_gain} \rfloor$. The evaluation artifact records split manifests, candidate workflow JSONs, transition-memory entries, retrieval logs, validation summaries, and command/run manifests.

C.3 Operational Definitions and Ablation Protocol

All rates are computed separately for each benchmark and then macro-averaged across the five benchmarks. Let $\mathcal{C}$ be the set of charged candidate attempts, $\mathcal{A} \subseteq \mathcal{C}$ the candidates that pass the complete acceptance gate, $\mathcal{G} \subseteq \mathcal{C}$ the candidates that reach guard evaluation, and $\mathcal{F} \subseteq \mathcal{C}$ the candidates whose applied workflows fail a runtime or smoke execution. An accepted transition is effective when it produces positive diagnostic gain without regressing any eligible non-target guard case. We report

$$\text{Eff.} = \frac{|\{c \in \mathcal{A} : \text{diag_gain}(c) > 0,\ \text{Regress}(c) = 0\}|}{|\mathcal{A}|}, \qquad \text{Accpt.} = \frac{|\mathcal{A}|}{|\mathcal{C}|}, \tag{20}$$

$$\text{Reg.} = \frac{\sum_{c \in \mathcal{G}} \text{Regress}(c)}{\sum_{c \in \mathcal{G}} |D^c_{\text{guard}}|}, \qquad \text{Fail} = \frac{|\mathcal{F}|}{|\mathcal{C}|}. \tag{21}$$

Here, $\text{Regress}(c)$ counts cases that are correct under the current accepted harness and incorrect under candidate c. Rates with an empty denominator are omitted from the corresponding macro-average.

For memory coverage, compatibility filtering is applied before similarity scoring. We compute cosine similarity between the L2-normalized embeddings of the current failure summary and each compatible stored failure summary using the fixed retriever encoder recorded in the run manifest, and assign an edit using its maximum score q. The fixed buckets are high for $q \ge 0.8$, medium for $0.6 \le q < 0.8$, low for $0.4 \le q < 0.6$, and no hit when no compatible entry exists or $q < 0.4$. Within a bucket, candidate share is the fraction of all proposed atomic edits assigned to it, adoption rate is the fraction of its edits that pass the complete acceptance gate, and non-target impact is the number of regressed guard cases divided by the number of eligible guard-case evaluations. A bucket with no candidate is reported as "–".

All ablations retain the same charged candidate-attempt budget, editor, data splits, and validation environment. Flat Accepted History replaces each typed accepted record with a chronological natural-language serialization of the same fields. GraphEdit+Gate w/o TM always supplies an empty-memory marker and performs no memory writes. Random Retrieval uniformly permutes the compatibility-filtered accepted entries while preserving their number and memory-context budget. If this pool is empty, it supplies the empty-memory marker. w/o Verified Write makes every schema-valid candidate that completes execution readable in later rounds, including candidates rejected by the gain or guard criteria. Static-invalid and runtime-incomplete candidates are excluded because they do not provide a complete transition record. Unconstrained Edit removes the typed single-scope edit restriction while preserving the shared candidate-attempt budget and validation gate.

C.4 Baseline Methods

This section defines the implementation scope of each baseline and the unified evaluation protocol.

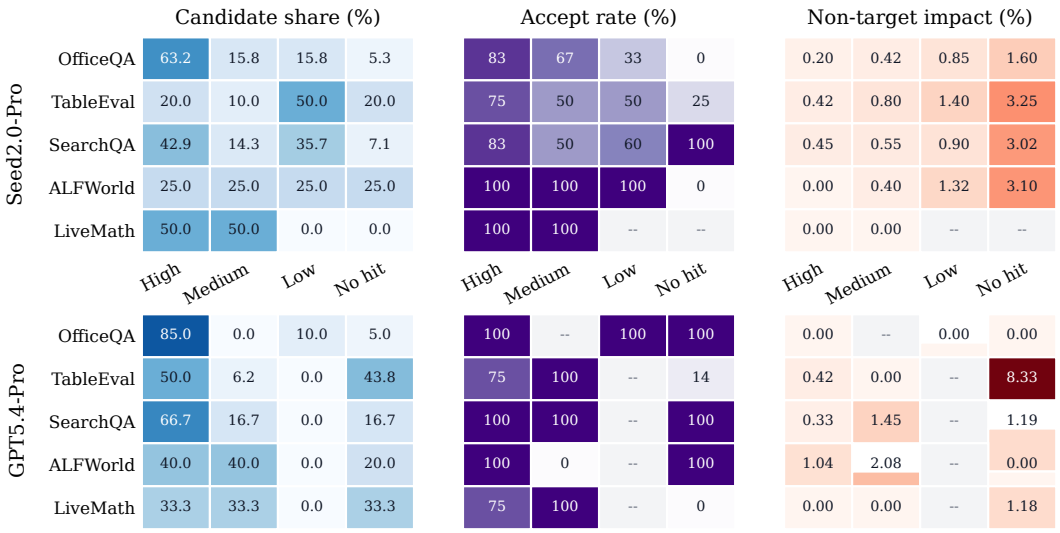


Figure 4: Memory-coverage buckets and candidate quality. Gray "–" denotes an empty bucket. Coverage is observational, whereas causal evidence comes from the controlled ablations.

C.4.1 Baseline scopes.

ReAct (Yao et al., 2022) is used as a non-optimizable trajectory-level baseline that interleaves reasoning and acting within each test instance. It does not use feedback from D_{train} or maintain persistent state across tasks.

GEPA (Agrawal et al., 2026) is used as a prompt optimization baseline. It leverages historical execution feedback from D_{train} to optimize prompts or node-level instructions while keeping the execution skeleton, tool interfaces, and deterministic code blocks fixed.

Trace2Skill (Ni et al., 2026) distills training trajectories into reusable skill documents, which are provided as external context to the target agent during inference. It does not access or modify the harness graph topology.

SkillOpt (Yang et al., 2026) treats skill documentation as a trainable external state of a frozen agent. An LLM optimizer translates scored rollout feedback into bounded textual edits, including addition, deletion, and replacement, and commits an update only when validation performance improves.

AFlow (Zhang et al., 2025a) and **Meta-Harness** (Lee et al., 2026) are the primary harness-level baselines. For AFlow, we implement its LLM-edited workflow graph search within the same charged candidate-attempt budget. Meta-Harness is a harness-code optimization baseline that uses training-side candidate harnesses, execution logs, and score feedback to revise retrieval, memory, prompt construction, and orchestration. Both baselines use the same diagnostic, guard, and final-reporting protocol as ConSeqHarness to ensure controlled structural search.

Baseline-specific adaptations. Concretely, GEPA was wrapped with an adapter that exposed only model-node prompts and node-local instructions as editable text components. Graph topology, tool interfaces, routing logic, deterministic code nodes, and the output parser were not exposed. Trace2Skill consumed executions on D_{train} after they were converted into its error and success analysis record schema. Its evolved skill directory was then supplied as inference-time context without changing the executable graph. SkillOpt consumed our split manifests and evaluator through its config-driven train and evaluation API, while keeping the optimized artifact as a skill document. AFlow received our benchmarks through its evaluator interface, allowing its native workflow operators to propose candidate workflows that were executed and scored by our task wrapper. Meta-Harness was initialized from the same smoke-checked W_0 and used its harness-code proposer to edit retrieval, memory, prompt construction, and orchestration code. These adaptations are only interface bindings. No baseline was given access to ConSeqHarness transition memory, typed edit records, or validation-gated memory writes.

C.5 Transition Diagnostics

Figure 4 reports these diagnostics. For each proposed atomic edit, we assign the maximum similarity to a compatible stored failure summary to the fixed coverage buckets in Appendix C.3. High-coverage candidates generally show higher adoption and lower non-target impact. For example, on TableEval with GPT5.4-Pro, high-coverage candidates have 75% adoption and 0.42% non-target impact, versus 14% and 8.33% for no-hit candidates. Because coverage is not randomly assigned, this result is descriptive rather than causal.

D Prompt and Skills

To improve reproducibility and clarify the operational details of our framework, we provide the prompts and skills used for harness generation, harness optimization, and modification guidance.

ConseqHarness Workflow Generation Protocol

The current workflow artifact is a **ConseqHarness graph JSON**, rather than the deprecated `workflow.yaml` DSL. Its top level is a mapping from node IDs to node dictionaries and may additionally contain `__meta__`. Each node field must satisfy the common node schema and the configuration requirements of its specific node type.

Local Tools.
The generation procedure uses the following local utilities:

- a structural workflow validator;
- a virtual executor for mock routing tests;
- a deterministic benchmark sampler for runtime smoke cases;
- a real single-case replay utility; and
- optional batch inference and metric-computation utilities.

Structural validation and virtual execution do not replace real single-case runtime replay.

External Inputs.

- A source document is required. Supported formats include Markdown, HTML, PDF, and image directories.
- The candidate feature pool is obtained from the key set of the first row in `offline_benchmark.jsonl`. This file must contain only the manifest-backed training split D_{train}, and its first row must be a valid JSON dictionary. The held-out D_{test} must never be supplied to the generation procedure.
- The complete training benchmark is also used for real-run acceptance. The generation process must not inspect only the first-row keys.
- Runtime smoke cases must be deterministically sampled from D_{train} according to labels, media inputs, and structured fields so that major terminal routes are covered.
- A runnable workflow from the same or a nearby domain may be used as a runtime-pattern reference. Its model parameters, multimodal input handling, terminal output format, and parallel-node style may be reused, but its domain-specific judgments must not be copied.

1. Entry Preconditions.

- The current task is workflow generation from a source document to a ConseqHarness workflow JSON.
- The required paths are `source_path`, `offline_benchmark_path`, and `workspace`.
- `offline_benchmark_path` must resolve to D_{train}; test examples are reserved for the single final evaluation after optimization.
- `domain` and `scene` provide task context only and must not be written to the top level of the workflow JSON.

2. Parameters.
Required parameters:

`source_path` Source document in Markdown, HTML, PDF, or image-directory form.

`offline_benchmark_path` Training benchmark JSONL path. The first-row key set defines the candidate feature pool, while the complete D_{train} file is used for runtime smoke testing. It must not contain held-out test examples.

`workspace` Working directory for prepared inputs, intermediate decomposition artifacts, workflow versions, outputs, and logs.

Conditionally required parameter:

`model_name` Required whenever the workflow contains a `PromptNode`. It must be a concrete model supported by the current runtime and must be written to every `PromptNode.node_config.model_name`. The final workflow must not retain a placeholder model name.

Optional parameters:

`workflow_name` Workflow name, generated by default from the final segment of `source_path`.

`max_nodes` Maximum number of nodes, with a default value of 40. It may be set to 0 only when the user explicitly requests an unlimited number of nodes.

`target_layers` Recommended number of processing layers, typically 2–3. Fixed `StartNode`, `ConclusionNode`, and `EndNode` wrapper layers are not counted.

`target_nodes_per_layer` Recommended workflow width, typically about three processing nodes per layer.

The procedure must not request or write `node_count`, `stage_count`, or `layer_count`. Node count and execution structure are naturally determined by graph nodes and their `children` edges. The parameters `max_nodes`, `target_layers`, and `target_nodes_per_layer` constrain generation complexity only and must not be written to the final workflow JSON.

3. Real Workflow JSON Constraints.
Every node must contain:

- `node_id`: a string equal to the corresponding top-level dictionary key;
- `node_type`: one of `StartNode`, `EndNode`, `ConclusionNode`, `PromptNode`, `CodeNode`, `ConditionNode`, `ParallelDivideNode`, `ParallelMergeNode`, `LoopBeginNode`, `LoopEndNode`, or `PlaceholderNode`;
- `node_name`: a unique name for non-system nodes, while system nodes use the fixed names `__start__`, `__conclusion__`, and `__end__`;
- `node_description`: a string describing the node;
- `node_config`: a dictionary containing exactly the configuration keys required by the node type;
- `input_fields`: a list whose references use the `<node_name>.<field>` format and may access nested fields such as `list_field[0].nested`;
- `output_fields`: a mapping whose values contain `type` and `description`, with optional `_meta_type` for complex list or dictionary fields; and
- `children`: a list of child node IDs.

The following runtime constraints must hold:

- There must be exactly one `StartNode` with `node_name="__start__"`.
- Multiple `EndNode`s are allowed. Their node name is fixed as `__end__`, and `node_config.output_map` must emit the final `workflow_result`.
- Multiple `ConclusionNode`s are allowed. Their node name is fixed as `__conclusion__`. The runtime terminal values may include `approve`, `reject`, `human`, and `unknown`. These values are runtime tokens and do not imply a particular application domain or binary classification task.
- `ConditionNode.node_config.conditions` must be an ordered list containing a fallback branch whose `condition_name="others"`.
- The set of all non-empty `condition_to` values in a `ConditionNode` must exactly equal its `children` set.
- Except for `ConditionNode` and `ParallelDivideNode`, each node may have at most one child.
- The graph must be reachable from the `StartNode`, connected as a whole, and free of illegal cycles.
- `ParallelMergeNode` and `EndNode` are special nodes that may be visited repeatedly.

Detailed field definitions and examples are provided in the released runtime schema specification.

4. Nine-Stage Generation Pipeline.
The stages are executed sequentially. Each stage consumes the previous artifact and writes an intermediate JSON or Markdown artifact under the workspace decomposition directory.

- **Stage 0: Source preparation.** When necessary, convert or organize the source document into a supported input format under the workspace input directory.
- **Stage 1: Information extraction.** Extract structured requirements, concepts, criteria, exceptions, and decision boundaries from the source document. Write the result to `extract_info.json`.
- **Stage 2: Atomic-task decomposition.** Decompose the extracted information into atomic judgments and routing relationships. Write the result to `atoms.json`.
- **Stage 3: Candidate feature extraction.** Read the first row of the offline benchmark and use its key set as the candidate input-feature pool. Write the result to `candidate_feature_pool.json`.
- **Stage 4: StartNode feature selection.** Select only the benchmark fields required by the workflow and write them to `features.json`.
- **Stage 5: Initial workflow generation.** Organize the atomic tasks, selected features, and routing relationships into a complete ConseqHarness graph JSON draft. Write the result to `workflow_draft.json`.
- **Stage 6: Workflow examination.** Check and correct the draft against the real node schema, graph-connectivity rules, input and output contracts, routing constraints, and terminal mappings. Write the corrected graph to `workflow_checked.json`.
- **Stage 7: Prompt and process refinement.** Refine each `PromptNode.prompt_template` and `process_fn` while preserving the checked graph structure. Write the result to `workflow_final.json`.
- **Stage 8: Structural validation.** Write the final graph to `versions/v0/workflow/workflow.json` and validate it with the workflow-checking utility. Validation failures enter the full-workflow rewrite procedure.
- **Stage 9: Real runtime smoke testing.** Deterministically sample representative cases from D_{train} and execute the final workflow with the real model runtime. Record the result in `runtime_smoke_summary.json`.

The workspace contains decomposition artifacts, prepared inputs, workflow versions, outputs, and logs. If generation resumes from an existing intermediate artifact, execution continues from the following stage rather than restarting from Stage 1.
The files `workflow_checked.json`, `workflow_final.json`, and the published `workflow.json` must each contain a directly loadable graph JSON body. Validation wrappers such as

```
\{has_errors, check_report, corrected_workflow_json\}
```

must not be written into workflow files. Any validation report must be stored separately.

5. Stage 8 and Stage 9 Validation.
Stage 8.1: Write the final workflow.
The final decomposition artifact is written to

`<workspace>/versions/v0/workflow/workflow.json`.

Prompt text must remain embedded in each `PromptNode.node_config.prompt_template`. A separate prompts directory must not be created.

Stage 8.2: Structural validation and repair.
The final workflow is validated using the released workflow-checking utility.

- Exit code 0 together with `Workflow OK!` indicates success.
- A non-zero exit code triggers a full-rewrite repair loop.
- Each repair round must rewrite the complete workflow JSON rather than applying partial patches.
- The default repair budget is six rounds followed by one final round.

- After six consecutive failures, the procedure returns to Stage 7, rewrites `workflow_final.json`, and performs one final complete workflow rewrite.
- Validation output is mapped to the released schema-repair checklist.
- Schema-shape errors must not immediately be escalated as missing user information.
- Escalation is permitted only when the documented repair escape conditions are satisfied, and validator output must be preserved.

Stage 8.3: Optional virtual dry run.
After structural validation succeeds, a manually constructed `StartNode` feature payload may be used to test workflow routing with mock `PromptNode` outputs. Mock outputs may be specified by either `node_id` or `node_name`.
The virtual executor is only a dry-run probe and does not replace Stage 9 real runtime replay. When parallel nodes are used, the virtual executor may inspect only a single mocked path. Complete parallel behavior must be validated structurally and through real runtime execution.

Stage 9: Required real runtime smoke test.
When D_{train} is available and the workflow contains at least one `PromptNode`, the procedure must:

1. deterministically sample smoke cases from D_{train} without hard-coding case IDs;
2. generate a label file for the sampled cases;
3. execute the final workflow on every sampled case using the real single-case runtime; and
4. write a runtime smoke-test summary.

The smoke test passes only when:

- each case process exits with code 0;
- the final `workflow_result` exists;
- the runtime `error_dict` is empty;
- no `PromptNode` JSON parsing error occurs;
- no `process_fn` field error occurs;
- no `ConditionNode` runtime error occurs;
- no multimodal input conversion error occurs; and
- no model-call parameter error occurs.

The workflow may be declared `publish_ready=true` only after both Stage 8 structural validation and Stage 9 real replay pass. If replay is blocked by missing dependencies, network restrictions, or unavailable model services, the result must be marked as a runtime-environment block. An untested workflow must not be described as runnable.

Stage 9.1: Preserve General Generation Experience.
Generation tasks should preserve reusable experience, but the experience artifact must not become a run log. Historical experience may be read before the extraction, decomposition, workflow-organization, or prompt-rewriting stages, but only de-identified and generalizable lessons may be reused.
Reusable generation experience may include:

- methods for decomposing source-document boundaries into atomic tasks and routing decisions;
- guidance on which judgments are better implemented by `PromptNode`, `CodeNode`, or `ConditionNode`;
- stable lessons for `StartNode` feature selection, `PromptNode` output contracts, and terminal output mapping;
- common structural and runtime failure types and their generalized repair strategies; and
- general lessons on avoiding overfitting, maintaining schema stability, and selecting representative smoke cases.

The experience artifact must not contain case IDs, smoke-case IDs, workflow version labels, candidate filenames, benchmark row IDs, concrete entity names, exact values, full source sentences, user-specific wording, workspace paths, per-run metric tables, or one-off implementation details.
Before writing an experience item, it should be abstracted into the following form:

```
Source boundary:
Generation failure mode:
Safer generalization:
Validation risk or smoke-test signal:
```

If a lesson cannot be expressed independently of a concrete sample, path, or workflow version, it must not be included in the reusable experience file.

6. Output Contract Returned to the Caller.
A successful generation task returns:

- `publish_ready=true`;
- the path of the final workflow JSON;
- a structural-validation exit code of 0;
- a concise summary of validator output;
- `runtime_smoke_test=passed`;
- the path of the sampled runtime smoke cases;
- the path of the runtime smoke summary; and
- the path of the reusable generation-experience artifact.

7. Failure Handling.

- **Stage 0 failure.** Request a source document in Markdown, HTML, PDF, or image-directory form.
- **Stage 3 failure.** Return `clarification_needed` if the benchmark path does not exist, the first row is invalid JSON, the first row is not a dictionary, or the dictionary contains no keys.
- **Stage 5, 6, or 7 invalid JSON.** Return to the preceding stage and rewrite the artifact while requiring one complete JSON object as output.
- **Stage 8 structural-validation failure.** Enter the complete-workflow rewrite loop rather than immediately returning `clarification_needed`.
- **Stage 9 runtime failure.** First distinguish workflow defects from runtime-environment blocks. Workflow defects must return to Stage 6 or Stage 7 for generalized repair, followed by repeated Stage 8 and Stage 9 validation. Repairs must not be specialized to a particular case ID.

8. Released Implementation.
The released generation skill contains:

- structural validation, virtual execution, deterministic smoke-case sampling, real inference, and metric scripts;
- specifications for supported source formats and workflow schemas;
- stage-specific instructions for source extraction, atomic-task decomposition, feature-pool construction, feature selection, graph organization, workflow checking, and prompt rewriting; and
- repair and runtime smoke-test guidelines.

These implementation files provide executable details, while the protocol above defines the generation procedure and its acceptance conditions.

Harness Three-Level Modification Guide

All harness edits must compile to `harness-schema.md` and must be serialized as a typed edit record

$$p_t = (\text{op}, \text{target}, \text{pre}, \delta, \text{post}, \text{scope}). \tag{22}$$

The edit must identify one affected node or path boundary, state its preconditions and postconditions, and preserve all non-target paths declared in `scope`. Multiple primitive changes may be grouped only when they share the same scope and pass the same validation gate.
Every edit must satisfy the following invariants:

- Use only supported `node_type` values and keep each `node_config` key set exact for its node type.

- Use `node_name`, rather than `node_id`, in `input_fields`, and reference only declared upstream outputs.
- Keep the target model, tool set, output parser, and terminal metric fixed throughout optimization.
- When adding or rewiring structure, synchronize `children`, `condition_to`, `branch_to`, `pair_node_id`, and all dependent input references.
- Preserve graph reachability, routing completeness, the default fallback path, and the required terminal `output_fields`.

The editor must choose the lowest effective level capable of addressing the current failure summary.

Level 1: Intra-node update.
Modify one existing node without changing graph topology. Permitted edits include local prompt instructions, few-shot examples, output constraints, node-local hyperparameters, and deterministic parsing or field-mapping logic inside a `CodeNode`. If a `PromptNode` must expose an intermediate decision or reasoning summary, add the corresponding field consistently to its output contract and to any downstream `input_fields`. The target model name and external tool interface remain unchanged.

Level 2: Local branching.
When a failure can be localized but cannot be corrected safely within one node, add a bounded remediation branch adjacent to the faulty node. First, make the upstream `PromptNode` or `CodeNode` emit a typed trigger field. Then insert a `ConditionNode` whose targeted branch handles the identified failure boundary and whose `others` branch preserves the original path. The condition list and `children` set must agree exactly, and the edit may affect only the declared local path.

Level 3: Terminal reflection.
When the failure arises from global answer consistency or terminal mapping, insert a bounded reflection, verification, or consensus step immediately before the terminal output. The new step may inspect declared upstream fields but must retain the original terminal schema and a fall-through route for non-target cases. All parent-child links, condition targets, terminal `input_fields`, and `EndNode.output_map` entries must be updated atomically.
A proposal that cannot be localized to one declared scope, violates the schema, changes the evaluation interface, or bypasses the fallback route is rejected before metric evaluation. A directly repairable schema consequence may receive one charged repair attempt; otherwise the candidate is discarded.

Optimization Protocol: Memory-Conditioned Sequential Harness Optimization

This protocol optimizes an executable ConSeqHarness workflow using training feedback while keeping the final test set strictly held out. It implements the memory-conditioned sequential process and validation gate described in Sections 3 and 4. Unless a benchmark manifest specifies otherwise, use data-sampling seed 40, all available training examples, and a $T = 18$ charged candidate-attempt budget.

1. Data Boundary and Objective
Let D_{train} and D_{test} be disjoint. Use the official training and validation portions when available. When a benchmark provides only a test pool, create a manifest-backed split with two thirds for D_{train} and one third for D_{test}; for small pools, use the fixed benchmark manifest. Store all selected example IDs. During optimization, run only on D_{train}. At round t:

- D^t_{diag} contains all examples currently answered incorrectly by W_t;
- D^t_{guard} contains all eligible current successes, previously fixed cases, and stable non-target cases; and
- D^t_{smoke} contains one target-path example and one non-target-path example when those path distinctions are available.

The test set must not be inspected by diagnosis, memory retrieval, candidate generation, validation, checkpoint selection, or stopping decisions. After optimization, freeze W_T and use D_{test} only for final held-out reporting.
For each task, the released evaluator maps the workflow output to binary success for that example. Report accuracy:

$$\text{ACC}(W, D) = \frac{100}{|D|} \sum_{x \in D} \mathbb{I}[\text{Eval}(W, x) = \text{correct}] \,. \tag{23}$$

Free-form answers and action sequences must be scored by the same task-specific evaluator and output parser for all methods. Missing, invalid, or out-of-domain terminal outputs count as incorrect.

2. Required Inputs
The optimization run requires:

- the initial smoke-checked workflow W_0, or the task specification needed by the generation protocol to produce it;
- manifest-backed D_{train} and D_{test};
- the task evaluator and terminal output parser;
- the fixed target model, tool set, and output-token cap;
- the GPT-5.5 editor configuration used to propose candidate edits;
- the workflow schema and the three-level modification guide;
- an output directory for workflows, traces, memory records, validation summaries, and run manifests; and
- the shared $T = 18$ charged candidate-attempt budget.

Resolve and record these inputs before optimization. Never infer an omitted label from model output, rewrite benchmark labels, or use test examples as demonstrations.

3. Reflection State and Transition Memory
At round t, maintain

$$s_t = (W_t, \xi_t, e_t, D^t_{\text{diag}}, D^t_{\text{guard}}), \qquad \bar{s}_t = (s_t, M_t), \tag{24}$$

where ξ_t and e_t contain the current training trajectories, failure clusters, and validation evidence. The long-term memory M_t is an append-only ledger of accepted transitions. Each entry records:

- the failure summary ϕ_i, including failure type, target node, trigger boundary, and stable dependency markers;
- the typed edit p_i, its declared scope, and workflow provenance;
- diagnostic gain, validation checks, and guard-set effects; and
- references to the workflow state before and after the accepted edit.

Retrieve only entries compatible with the current task mode, workflow schema, output contract, evaluator, and target configuration. Filter compatible entries by overlapping failure type and target location, then prioritize larger diagnostic gains and smaller guard effects. If no entry is compatible, supply an explicit empty-memory marker to the editor.
Rejected candidates must not enter M_t. Their failure summaries may be used only for short-term candidate deduplication or the current-round risk record. No memory entry may contain information derived from D_{test}.

4. Canonical Optimization Loop
For $t = 0, \ldots, T-1$, execute the following steps in order:

1. **Refresh the training state.** Execute W_t on all of D_{train} with the fixed target model, tools, parser, and token cap. Reconstruct D^t_{diag}, D^t_{guard}, and D^t_{smoke} from the stored manifest policy. If D^t_{diag} is empty, carry W_t forward as the final checkpoint and stop optimization.

2. **Diagnose.** Inspect traces for every current failure and cluster failures by reusable execution cause. The summary $\phi_t = \Phi(e_t)$ must identify the failing node or path, relevant evidence, trigger boundary, expected correction, and non-target paths at risk. Example IDs are evidence for diagnosis and must never become literal edit conditions.

3. **Read memory.** Retrieve the compatible, previously accepted transitions from M_t using the provenance and attribute filters in item 3. Provide their edit scopes and validation records to the editor together with ϕ_t. If retrieval is empty, propose from the current state and failure summary alone.

4. **Propose a constrained edit.** The editor emits $p_t = (\mathrm{op}, \mathrm{target}, \mathrm{pre}, \delta, \mathrm{post}, \mathrm{scope})$ and chooses the lowest effective level from the three-level guide. Each proposal must localize to one affected node or path boundary. Multiple primitive changes may be committed together only when they share that scope.

5. **Apply and check static legality.** Apply the proposal to a copy of W_t. Check node schemas, field references, graph reachability, route completeness, tool signatures, and terminal mappings. Invalid candidates are rejected before metric evaluation. A directly repairable schema consequence may receive one charged repair call and then must re-enter static validation.

6. **Run path-level smoke tests.** For structural, routing, code, or terminal changes, execute the target and non-target smoke paths in D^t_{smoke}. Require valid termination, an empty runtime error record, and a parser-valid terminal output.

7. **Measure diagnostic gain.** Evaluate the candidate W' on all of D^t_{diag}. Let $\mathrm{diag_gain}_t$ be the number of current failures corrected by W'. Require

$$\mathrm{diag_gain}_t \geq \Delta^t_{\min} = \max\big(1, \lceil 0.03 |D^t_{\mathrm{diag}}| \rceil\big) . \tag{25}$$

8. **Control guard regression.** Evaluate W' on all of D^t_{guard}. Let $\mathrm{Regress}_t$ be the number of guard examples correct under W_t but incorrect under W'. Require both

$$\mathrm{Regress}_t \leq \lceil 0.2 |D^t_{\mathrm{guard}}| \rceil , \qquad \mathrm{Regress}_t \leq \lfloor 0.5 \, \mathrm{diag_gain}_t \rfloor . \tag{26}$$

9. **Commit or roll back.** Accept W' only if static validation, runtime smoke testing, diagnostic gain, and both guard constraints pass. For an accepted candidate, set $W_{t+1} = W'$ and append its complete validation certificate to M_{t+1}. For a rejected candidate, keep $W_{t+1} = W_t$ and $M_{t+1} = M_t$. At most one state update may be accepted in each round.

10. **Refresh after acceptance.** After an accepted update, rerun the current workflow on all of D_{train}, rebuild the next diagnostic and guard sets, and store their manifests before the next proposal.

Candidates are evaluated sequentially. Every optimization attempt that proposes or revises a workflow counts against the shared budget, whether the candidate is accepted, rejected, repaired, or regenerated. Infrastructure retries that leave the candidate and validation sets unchanged are logged but do not count as new optimization attempts.

5. Acceptance Predicate

The operational gate is:

$$\mathbb{I}_{\mathrm{accept}} = \mathrm{StaticPass}(W') \cdot \mathrm{RuntimePass}(W', D^t_{\mathrm{smoke}}) \tag{27}$$

$$\cdot \, \mathbb{I}\big[\mathrm{diag_gain}_t \geq \Delta^t_{\min}\big] \cdot \mathbb{I}\big[\mathrm{Regress}_t \leq \lceil 0.2 |D^t_{\mathrm{guard}}| \rceil\big] \tag{28}$$

$$\cdot \, \mathbb{I}[\mathrm{Regress}_t \leq \lfloor 0.5 \, \mathrm{diag_gain}_t \rfloor] . \tag{29}$$

Only a candidate with $\mathbb{I}_{\mathrm{accept}} = 1$ may change the current harness or enter long-term transition memory. Diagnostic accuracy, smoke results, and guard effects are training-side validation signals, not test metrics.

6. Shared Baseline and Cost Protocol

All optimizable methods use the same D_{train}, D_{test}, target model, task tools, evaluator, output parser, output-token cap, and the same $T = 18$ charged candidate-attempt budget. Methods that accept an

initial executable harness start from the same smoke-checked W_0. Non-structural baselines keep the graph topology, tool interfaces, deterministic code, and terminal parser fixed according to their method scope. Structural baselines may modify only the components permitted by their published optimization interface.

Charge every candidate proposal, revision, and directly repairable schema repair to the shared candidate-attempt budget. Record editor calls, target-model executions, validation executions, and infrastructure retries separately. Memory retrieval, candidate generation, and validation occur only during offline optimization and are excluded from the deployment path of W_T.

7. Artifacts and Logging

For every run, store:

- the resolved task, model, tool, evaluator, parser, token-cap, seed, and budget configuration;
- the fixed train/test split manifest and every round-specific diagnostic, guard, and smoke manifest;
- W_0, every candidate workflow, every accepted checkpoint, and the final W_T;
- transition-memory entries, retrieval results, failure summaries, static-check reports, smoke traces, diagnostic gains, and guard effects;
- candidate acceptance, repair, rejection, and infrastructure-retry records; and
- the exact commands and run manifest needed to reproduce the run.

The optimization log must keep training-side validation results separate from the final held-out result. Do not cache, retrieve, or expose test-set predictions before the final workflow is frozen.

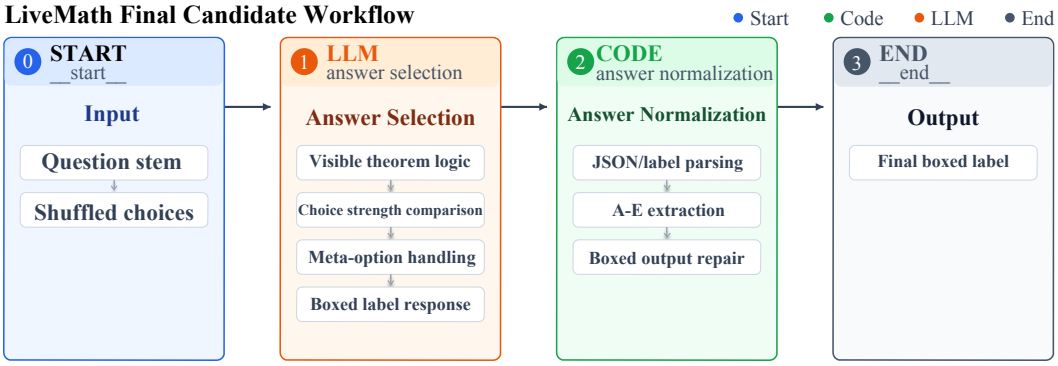
8. Final Evaluation and Reporting

Stop when the $T = 18$ charged candidate-attempt budget is exhausted, all training cases are solved, or no admissible candidate remains. If training cases are solved early, carry the last accepted harness forward to W_{18}.

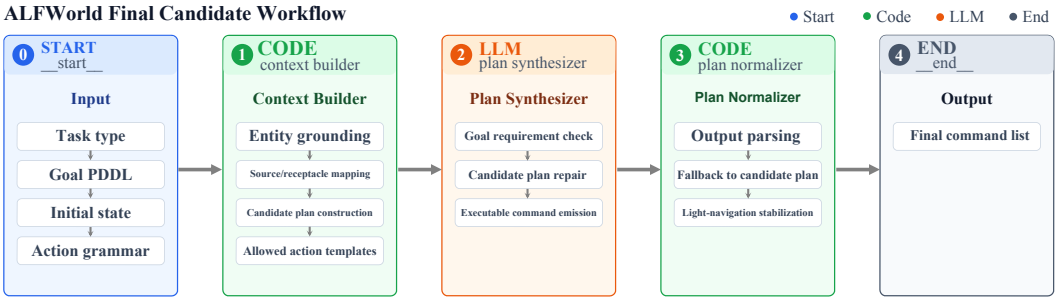
After stopping, freeze the final harness and evaluate it once on D_{test}. Report per-benchmark ACC and the arithmetic mean across the five benchmarks. Also report the retained edit scope, longest deployment-path LLM-node count, output-token cap, and non-LLM operations. The final test result must not be used to select, repair, or roll back a checkpoint.

E EXAMPLES OF OPTIMIZED HARNESSES

Figure 5 presents representative examples of harnesses optimized by ConSeqHarness, with the LiveMath and ALFWorld workflows shown in Figures 5a and 5b, respectively. These examples show how ConSeqHarness adapts executable harness structures to task-specific requirements while preserving the tool interfaces, input–output specifications, and evaluation protocol used in the main experiments.



(a) Example harness for LiveMath. The optimized harness incorporates problem decomposition, solution verification, and final-answer normalization to improve mathematical reasoning reliability.



(b) Example harness for ALFWorld. The optimized harness adds explicit state tracking and action validation modules to stabilize long-horizon decision making in embodied task environments.

Figure 5: Example harness for two benchmarks.