EVOUNDO: RECOVERABILITY-CONSTRAINED SELF-EVOLUTION FOR LLM AGENT HARNESSES

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM agents increasingly modify their own prompts, tools, middleware, resources, and execution harnesses at runtime. Such self-evolution can improve capability, but a successful mutation may leave persistent effects that cannot be safely reversed in states different from the one in which it was created. We introduce **EvoUndo**, a framework for representing, synthesizing, diagnosing, and independently verifying recoverability of model-generated self-modifications across counterfactual states. Across 600 unseen self-evolution tasks using single-sample zero-shot generation, we identify 197 capability-improving mutations that fail recoverability verification. Under the original recovery representation, conventional repair strategies recover 0/197 of these natural failures. Deterministic oracle analysis recovers 48/197 under the original recovery language L_0, while the extended recovery calculus increases empirical oracle recovery to 191/197. A protocol-locked 2×2 grounding-by-expressivity intervention then separates two bottlenecks: state-grounded diagnostic feedback increases successful recovery from 0/48 to 38/48 (79.2%) when the original language is sufficient, while extending the recovery language enables recovery on 142/143 (99.3%) failures in the oracle-defined S_1 stratum. On the primary gpt-oss-120b backbone, adding state-grounded diagnostics to the richer language reduces recovery to 133/143 (93.0%); the negative interaction did not reproduce under the Qwen3.8-27B constrained-decoding replication configuration, which preserves the directions of the grounding and expressivity effects. These results indicate that reliable agent self-evolution requires co-designing verification, state grounding, witness semantics, and recovery-language expressivity rather than relying on iterative prompting alone.

1 INTRODUCTION

LLM agents are increasingly becoming *mutable systems*. Beyond selecting discrete actions, modern agents can modify prompts, install or remove tools, change middleware, alter configurations, allocate resources, register event listeners, and reconfigure their execution harnesses at runtime (Yin et al., 2024; Zhang et al., 2026b; Lin et al., 2026). These capabilities create a path toward *self-evolving agent harnesses*: systems that autonomously adapt their execution environment to novel tasks and failure modes.

Most self-improvement procedures naturally optimize the forward effect of a modification: given a harness state H and a candidate mutation m, a candidate is favored whenever it improves a task objective, $J(m(H)) > J(H)$. However, forward improvement is insufficient for long-lived autonomous systems. A successful mutation may overwrite configuration values, reorder middleware chains, shadow existing tools, create temporary files, or leak background resources. If that mutation later becomes obsolete, causes a regression, or conflicts with subsequent updates, simply applying a static inverse operation may fail to restore the prior state. Crucially, the correct recovery procedure is often *state-dependent*: restoring observational equivalence requires information that existed in the pre-mutation state.

This motivates a fundamental requirement for autonomous self-evolution: capability-improving mutations must also be *robustly recoverable*. After applying a mutation to state s, the system must be able to utilize pre-mutation state evidence to restore a state observationally equivalent to s. Further-

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.

more, this property must hold not merely in the nominal state used during construction, but across counterfactual states in which the mutation may operate.

We formalize and investigate this challenge: *How should an autonomous agent search for capability-improving harness mutations when robust recoverability is an explicit invariant of the search space?*

We introduce **EvoUndo**, a framework for *recoverability-constrained self-evolution*. EvoUndo provides a framework for representing, synthesizing, diagnosing, and independently verifying recoverability of model-generated self-modifications. Candidates are executed across counterfactual harness states and admitted only when the recovered states satisfy typed observational equivalence with their corresponding pre-states. Recovery failure is treated as an actionable defect to diagnose and repair before a mutation is persistently admitted.

A central insight of this work is that recoverability is *relational*: whether a mutation is recoverable depends jointly on the state distribution, witness capture semantics, observational contract, and expressivity of the recovery language. On 120 deliberately corrupted but recoverable mutations, verifier-guided typed diagnosis repairs 114/120 (95.0%) cases at budget four, compared with 4/120 (3.3%) under independent regeneration. In sharp contrast, across 600 unseen self-evolution tasks using single-sample zero-shot generation, 197 capability-positive mutations fail recovery verification, and conventional verifier-guided repair recovers 0/197 under the original recovery representation.

A zero-generation solvability audit reveals that only 48/197 natural failures are recoverable by the deterministic oracle under the original language L_0, whereas an extended language L_1 raises empirical oracle recoverability to 191/197. A protocol-locked 2×2 grounding-by-expressivity factorial separates these failure modes: state-grounded diagnostic feedback raises recovery from 0/48 to 38/48 (79.2%) when L_0 is sufficient, while L_1 unlocks recovery on 142/143 (99.3%) failures in the oracle-defined S_1 stratum. On gpt-oss-120b, adding state-grounded diagnosis on top of L_1 reduces recovery to 133/143 (93.0%), while this negative interaction did not reproduce under the Qwen3.8-27B constrained-decoding replication configuration.

Contributions.

1. We formalize **recoverability-constrained self-evolution** and introduce **EvoUndo**, coupling harness mutations with witness capture, counterfactual verification, typed diagnosis, and closed-loop recovery synthesis.
2. We establish a natural failure benchmark from 600 self-evolution tasks, identifying 197 capability-positive recovery failures where standard iterative repair achieves 0/197 success under the base recovery representation.
3. We decouple natural recovery failures into **grounding** and **expressivity** bottlenecks: state-grounded diagnostic feedback recovers 38/48 (79.2%) failures in S_0, while an extended recovery calculus enables recovery on 142/143 (99.3%) failures in S_1.
4. We observe a non-monotonic interaction on the primary configuration between diagnostic granularity and recovery-language capacity: state-grounded feedback degrades recovery under L_1 on gpt-oss-120b, while this negative interaction did not reproduce under the Qwen3.8-27B constrained-decoding replication configuration.

2 Problem Formulation

2.1 Self-Evolving Agent Harnesses

We model an agent harness as $H = (s, \Pi)$, where $s \in \mathcal{S}$ is a concrete persistent harness state and $\mathcal{S}$ is the state space. The state includes configuration, prompt templates, routing, tool registries, middleware sequences, event-listener bindings, sandboxed files, and managed resources; Π is the execution policy. A mutation has a state projection $m : \mathcal{S} \to \mathcal{S}$. At a nominal state $s_\star \in \mathcal{S}$, it is *capability-improving* when $\Delta J(m; s_\star) = J(m(s_\star)) - J(s_\star) > 0$ for a forward objective J. We reserve S_0, S_1 for the empirical task strata below. Persistent modifications must carry verified recovery semantics before permanent admission.

2.2 WITNESSED RECOVERY AND EXAMPLE

Because forward mutations may overwrite information, $m(s)$ is generally insufficient to reconstruct its predecessor. The pre-state capture program $w : \mathcal{S} \to \mathcal{W}_{\text{pre}}$ records recovery-relevant prior values and existence flags. Forward execution additionally produces an allocation receipt:

$$[\![m]\!](s) = (m(s), \mathcal{R}_m(s)), \qquad \hat{s} = u(m(s), w(s), \mathcal{R}_m(s)).$$

Here $\mathcal{R}_m(s) \in \mathcal{P}(\mathcal{K}_{\text{res}} \times \mathcal{H})$ binds symbolic resource keys to dynamically acquired handles; u consumes both the pre-state witness and this mutation-produced receipt. A pre-state witness cannot contain a future socket handle. Resource capture stores lightweight allocation descriptors, not serialized live sockets; symbolic recovery operations use the runtime registry to close newly allocated handles and remove bindings.

Example: Consider a configuration mutation $m =$ `set_config("timeout_sec", 60)` applied to an initial state s where `config["timeout_sec"] = 30`. Because $m(s)$ contains only 60, reconstructing s requires witness capture $w(s)$ recording pre-state value 30 and that the key existed. The recovery program u executes `restore_config("timeout_sec", witness_value)`, successfully restoring 30. If the key did not previously exist, recovery instead deletes the newly introduced binding. In contrast, consider inserting a rate limiter into a middleware chain: m inserts a filter at index 0. Here, a simple scalar inverse is insufficient; recovery must capture pre-mutation sequence positions and execute ordered indexed restoration (L_1), illustrating why richer state surfaces demand structured recovery calculi.

2.3 TYPED OBSERVATIONAL EQUIVALENCE

Rather than demanding full byte-level equality over an entire operating environment, recovery is evaluated using state-specific *observational equivalence* $\hat{s} \simeq_{\mathcal{C}_e} s$ governed by an effect contract $\mathcal{C}_e$. The contract declares typed state targets whose effects must be covered. Semantic representations are recursively canonicalized: dictionary keys are sorted, volatile identifiers/pointers are mapped to canonical placeholders, and file contents are verified via SHA-256 hashes (excluding ephemeral timestamps/inodes). Configuration mappings, tool parameter schemas, middleware ordering, listener callbacks, and resource descriptors are checked against their pre-mutation semantic definitions (details in Appendix D).

Let Σ denote the modeled persistent state surfaces. Typed restoration rules apply only within $\mathcal{C}_e$; outside it, the verifier requires strict canonical equality. Writing $\hat{s} \simeq_{\mathcal{C}_e} s$ for this combined relation,

$$\begin{aligned} s_{\text{rec}} \equiv_{\mathcal{T}(\mathcal{C}_e)} s_{\text{pre}} \iff & (\forall \sigma \in \mathcal{C}_e,\ s_{\text{rec}}[\sigma] \equiv_\sigma s_{\text{pre}}[\sigma]) \\ & \wedge (\forall \sigma \in \Sigma \setminus \mathcal{C}_e,\ s_{\text{rec}}[\sigma] = s_{\text{pre}}[\sigma])\,. \end{aligned} \tag{1}$$

Thus persistent recovery corruption outside $\mathcal{C}_e$ fails verification. An empty contract requires global canonical equality and cannot cover any nonempty forward effect $E(m, s)$. These checks concern final modeled persistent state: transient recovery-side effects that disappear before final-state verification are not independently constrained.

2.4 COUNTERFACTUAL RECOVERABILITY

Recovery must generalize beyond the specific state on which a mutation was drafted. Let $\mathcal{L}$ denote the available recovery language and Q a distribution over counterfactual harness states. We define counterfactual recoverability as:

$$\mathcal{R}_{\mathcal{L},Q,\mathcal{C}_e}(m) = \mathbf{1}\left[\exists (w, u) \in \mathcal{L} : \Pr_{s \sim Q}\left[u(m(s), w(s), \mathcal{R}_m(s)) \simeq_{\mathcal{C}_e} s \ \wedge\ E(m, s) \subseteq \mathcal{C}_e\right] \geq \tau_R\right], \tag{2}$$

where $\tau_R \in (0, 1]$ is the required recovery threshold. Recoverability is therefore an inherently *relational* property parameterized by the forward mutation, witness/recovery representations, observational contract, and state distribution Q.

2.5 RECOVERABILITY-CONSTRAINED OBJECTIVE

The goal of recoverability-constrained self-evolution is to discover forward-improving mutations that satisfy robust counterfactual recovery:

$$\begin{aligned} \max_{m,w,u,\mathcal{C}_e} \; & J(m(s_\star)) \\ \text{s.t. } & \Delta J(m; s_\star) > 0, \quad (w,u) \in \mathcal{L}, \quad \mathcal{C}_e \text{ valid}, \\ & \Pr_{s\sim Q}[\hat{s} \simeq_{\mathcal{C}_e} s \;\wedge\; E(m,s) \subseteq \mathcal{C}_e] \geq \tau_R. \end{aligned} \tag{3}$$

Our recovery-repair experiments hold $m = m_0$ fixed and restrict search to $(w, u, \mathcal{C}_e)$. This intentional control isolates witness construction, recovery synthesis, and effect-contract reasoning while preventing apparent recovery gains from weakening forward capability. It also limits the study: joint optimization of forward mutation and recovery was not evaluated. Future work could explore capability-equivalent but more reversible forward mutations.

3 EVOUNDO

3.1 CANDIDATE REPRESENTATION AND EFFECT CONTRACTS

An EvoUndo candidate is a 4-tuple $\xi = (m, w, u, \mathcal{C}_e)$, where m is the forward mutation, w is the witness program, u is the recovery program, and $\mathcal{C}_e$ is the declared effect contract. In EvoUndo recovery repair, the forward mutation m is strictly immutable ($m_{\text{repaired}} \equiv m_0$): m_0 is locked at Turn 0 and executed verbatim. Only the witness capture program w, recovery program u, and declared effect contract $\mathcal{C}_e$ may be revised across repair turns. The contract is constrained to the runtime's typed effect-contract vocabulary; if omitted, the compiler infers it from the locked forward operations. Although mutable, $\mathcal{C}_e$ is not trusted as complete: the runtime independently computes $E(m, s) = \text{SnapshotDiff}(s, m(s)) \cup \text{ExecutionTraceEffects}(m, s)$ and admission requires $E(m, s) \subseteq \mathcal{C}_e$. Omitting an observed effect therefore fails the contract audit, whereas overdeclaration enlarges the equivalence scope that recovery must satisfy. The immutability of m prevents evasion by weakening the forward mutation to a no-op, while independent effect tracking prevents evasion by shrinking the declared recovery scope.

3.2 COUNTERFACTUAL ROUND-TRIP VERIFICATION

For each counterfactual state $s \in \mathcal{Q} = \mathcal{Q}_{\text{dev}} \cup \mathcal{Q}_{\text{hid}}$, EvoUndo evaluates the round trip $w_{\text{pre}} = w(s)$, $(s', \mathcal{R}_m(s)) = [\![m]\!](s)$, and $\hat{s} = u(s', w_{\text{pre}}, \mathcal{R}_m(s))$, checking whether $\hat{s} \simeq_{\mathcal{C}_e} s$. Development states Q_{dev} generate state-level diagnostic feedback. In the original factorial, hidden state contents and per-state outcomes are withheld, although aggregate IID/OOD recovery LCB summaries are returned during repair. The fresh-holdout study instead uses a fully feedback-isolated hidden set for final admission.

3.3 DIAGNOSTIC FEEDBACK GRANULARITY

When a development round trip fails, the verifier computes the state residual $\Delta_{\text{res}} = \text{Diff}_{\mathcal{C}_e}(s, \hat{s})$. Under **coarse typed diagnosis** (D_0), the model receives the affected subsystem and defect classification without exact residual locations. Under **state-grounded diagnosis** (D_1), the feedback supplies a composite bundle of canonical state addresses, forward effect ordering, observed residuals, residual counts, and verifier observations (prompts in Appendix H). Causal interpretation applies to this bundle as a whole; addresses were not isolated from the other diagnostic fields.

3.4 RECOVERY LANGUAGES

Synthesis is bounded by a formal recovery language $\mathcal{L}$. Table 1 gives the implemented primitives and operational semantics. Witness capture runs before m, recording typed pre-state values and existence flags in $w(s)$; recovery also consumes mutation-produced resource receipts. A primitive restores the previous binding if it existed, or deletes/unregisters a newly introduced binding if it did not. L_0 covers configuration, prompt, tool-registry, and routing state. L_1 extends it with middleware

sequences, event listeners, sandboxed files, managed sockets, and ordered composition of recovery operations across surfaces. For example, a mutation that prepends middleware and registers a listener is recovered by unregistering/restoring the listener before restoring the prior middleware sequence in strict Last-In, First-Out (LIFO) order relative to the forward operations. The runtime rejects any L_1-only primitive emitted under L_0 and consumes that synthesis attempt. These are implemented operational semantics, not a completeness characterization of recovery programs.

Table 1: Recovery language calculi (L_0 and L_1) over modeled persistent state targets.

Language	Stage	Primitive(s)	Target / state role
L_0	Capture	`capture_config`, `capture_prompt`, `capture_tool`, `capture_routing`	Typed pre-state witness
L_0	Forward	`set_config`, `set_prompt`, `register_tool`, `set_routing`	Locked forward effect
L_0	Recovery	`restore_config`, `delete_config`, `restore_prompt`, `restore_tool`, `unregister_tool`, `restore_routing`	Restore or remove prior binding
L_1	Capture	`capture_middleware`, `capture_listener`, `capture_file`, `capture_socket`	Structured pre-state witness
L_1	Forward	`add_middleware`, `add_listener`, `write_file`, `allocate_socket`	Locked structural effect
L_1	Recovery	`restore_middleware`, `restore_listener`, `unregister_listener`, `restore_file`, `delete_file`, `release_socket`	Structured restoration or removal

3.5 Closed-Loop Synthesis and Admission

If a capability-positive mutation fails recovery verification, EvoUndo initiates bounded synthesis over $(w, u, \mathcal{C}_e)$ up to budget $B \in \{1, 2, 3, 4\}$ with early stopping while keeping m_0 fixed. Proposals passing $\mathcal{Q}_{\text{dev}}$ are evaluated on $\mathcal{Q}_{\text{hid}}$. The executed primary protocol uses separate IID and OOD hidden splits ($n = 20$ each). Each independently requires the Wald normal-approximation lower endpoint $\max(0, \hat{p} - 1.96\sqrt{\hat{p}(1-\hat{p})/n}) \geq \tau_R = 0.85$, where $\hat{p} = k/n$. Thus 20/20 and 19/20 pass, while 18/20 fails: at least 19/20 successes are required on each split. Post-hoc pooled $N = 40$ two-sided 95% Wilson rescoring at threshold 0.85 gives identical decisions for all recorded candidate evaluations; it was not the executed protocol (Appendix J.1). EvoUndo fails closed: syntax errors, unhandled exceptions, or out-of-language primitives result in immediate rejection, ensuring every admitted state transition satisfies the configured empirical counterfactual recoverability criterion over the modeled harness state.

4 Experimental Setup

4.1 Harness State and Task Benchmark

The natural benchmark comprises six 100-task architectural families ($N = 600$): **Configuration**, **Tools**, **Middleware**, **Listeners**, **Resources**, and **Multi-Surface**. Multi-Surface is a composite benchmark family, not an atomic harness surface. These evaluation groupings do not enumerate persistent state structures one-to-one: prompt and routing state are also modeled; Resources covers file- and resource-backed mutations; and Multi-Surface tasks combine mutations touching at least two persistent state targets. Forward capability $J_t(m(s_\star))$ evaluates postcondition satisfaction for each task t independently of recovery syntax. Config has zero recovery-defective tasks in the frozen primary cohort (Table 6).

4.2 Model, Inference, and Verification Protocol

Factorial repair and synthesis experiments use the open-weights `gpt-oss-120b` model under native MXFP4 precision at temperature 0.2 with medium reasoning effort on eight NVIDIA H200 GPUs. In the primary factorial study on the frozen 197-task cohort, each candidate is evaluated

across $|\mathcal{Q}_{\text{dev}}| = 10$ development states and $|\mathcal{Q}_{\text{hid}}| = 40$ hidden states (20 IID, 20 OOD), gated independently by the split-Wald rule in Section 3. IID states perturb pre-state boundary values, prior-existence states (absent vs. conflicting tool schemas), and combinatorial surface assignments. OOD states introduce true structural shifts: unfamiliar nested keys, conflicting socket allocations, prepended high-priority middleware, multi-level file paths, and reverse routing defaults (detailed taxonomy in Appendix D). Generalization is further verified on an untouched 300-task fresh hold-out with 100-state hidden evaluation (Section 7). In a separate controlled audit of 300 intentionally incomplete effect contracts across all six harness families, independent dynamic effect tracking detected every injected undeclared semantic effect (300/300), with no false positives on 300 matched complete contracts; Appendix D provides details.

4.3 Controlled Benchmark and Natural Failure Bank

We construct a **controlled benchmark** of 120 capability-positive mutations with injected witness/recovery defects and a **natural failure bank** by single-sample zero-shot generation across 600 tasks. Of these, 281 are admissible, 122 are capability-inadequate, and **197** are capability-positive but fail recovery verification, forming our frozen primary cohort. Turn-0 generation used one candidate per task, zero demonstrations, and no rerolls; complete generation parameters are reported in Appendix F.

4.4 Solvability Audit and Mechanistic Strata

To distinguish synthesis search failures from empirical representational limits under the implemented recovery languages, we perform a zero-generation constructive oracle audit. Under base language L_0, only 48/197 mutations are oracle-recoverable. Under extended language L_1, 191/197 are oracle-recoverable. We define empirical oracle recoverability $\mathcal{R}^{\text{oracle}}_{\mathcal{L},Q}(m) \in \{0,1\}$ as successful recovery synthesis and hidden-state admission by the deterministic constructive oracle under language $\mathcal{L}$. This partitions the failure bank into two disjoint mechanistic strata:

$$S_0 = \{m : \mathcal{R}^{\text{oracle}}_{L_0,Q}(m) = 1\}, \quad |S_0| = 48, \tag{4}$$

$$S_1 = \{m : \mathcal{R}^{\text{oracle}}_{L_1,Q}(m) = 1 \wedge \mathcal{R}^{\text{oracle}}_{L_0,Q}(m) = 0\}, \quad |S_1| = 143. \tag{5}$$

Six tasks remain unrecovered because of implemented oracle heuristics rather than demonstrated theoretical impossibility and are excluded from stratum contrasts (Appendix G.4). The oracle uses privileged pre-state and evaluator information only for retrospective stratification; no model receives its programs or hidden outcomes. It constructs its own tight effect contract from forward-operation surfaces and observed mutation effects, independently of candidate declarations. Stratum membership was unchanged across 10 independent resamplings from the implemented counterfactual generator (Appendix G.6).

4.5 Grounding–Expressivity Factorial and Contrasts

We evaluate a 2×2 factorial crossing diagnostic granularity with recovery language: $\{D_0, D_1\} \times \{L_0, L_1\}$. Before execution, we protocol-locked three confirmatory paired contrasts:

- C_1: D_1L_0 vs. D_0L_0 on S_0 (tests state-grounded diagnostic feedback when L_0 is sufficient).
- C_2: D_0L_1 vs. D_0L_0 on S_1 (tests recovery-language expressivity on the S_1 stratum).
- C_3: D_1L_1 vs. D_0L_1 on S_1 (tests diagnostic grounding on top of an expressive language).

We report Rescue@B (the fraction of recovery-defective tasks successfully admitted within at most B sequential synthesis turns, stopping early on success; $B = 4$ in the primary factorial), paired risk differences Δ with 95% bootstrap CIs, and exact two-sided McNemar tests with Holm–Bonferroni correction. Extended related work is detailed in Appendix A.

Table 2: Final protocol-locked Grounding–Expressivity Factorial results on the frozen natural failure cohort ($N = 197$) at budget $B = 4$. S_0 ($N = 48$) denotes failures recovered by the deterministic oracle under L_0; S_1 ($N = 143$) denotes failures recovered under L_1 but not by the L_0 oracle.

Factorial Condition	**Full Cohort** ($N = 197$)	**Stratum** S_0 ($N = 48$)	**Stratum** S_1 ($N = 143$)
D_0L_0 (Coarse Diagnosis, Base Language)	0/197 (0.0%)	0/48 (0.0%)	0/143 (0.0%)
D_1L_0 (Grounded Bundle, Base Language)	38/197 (19.3%)	38/48 (79.2%)	0/143 (0.0%)
D_0L_1 (Coarse Diagnosis, Rich Language)	**180/197 (91.4%)**	38/48 (79.2%)	**142/143 (99.3%)**
D_1L_1 (Grounded Bundle, Rich Language)	167/197 (84.8%)	34/48 (70.8%)	133/143 (93.0%)

5 Controlled and Natural Recovery Results

5.1 Controlled Recoverability: Repair Succeeds When Representation Aligns

On the 120-task controlled benchmark, independent regeneration recovers only 4/120 (3.3%) defects at budget $B = 4$. In contrast, closed-loop verifier feedback achieves high repair rates: generic feedback recovers 104/120 (86.7%), raw verifier traces recover 101/120 (84.2%), typed diagnosis recovers 114/120 (95.0%), and prescriptive hints reach 117/120 (97.5%). Thus, when defects fall within the language's representational envelope, closed-loop repair is highly effective.

5.2 Natural Failures Break Conventional Repair

On the frozen cohort of 197 natural self-evolution failures, this success collapses entirely. Under the base representation (L_0), all four verifier-guided repair modes achieve **0/197 (0.0%)** recovery at $B = 4$, while independent regeneration recovers 6/197 (3.0%). Baseline taxonomy classification identified 180/197 natural failures outside the predefined controlled defect taxonomy. This set is distinct from the 180/197 tasks rescued by D_0L_1: the overlap is 164 tasks, with 16 taxonomy-only and 16 rescue-only tasks. Equal cardinalities do not identify the same tasks.

End-to-end 600-task accounting. Of all 600 unseen tasks, 478/600 (79.67%) produced capability-positive mutations: 281 were already admissible and 197 were recovery-defective (197/478, 41.2% of capability-positive mutations). Under D_0L_1 at $B = 4$, EvoUndo rescues 180/197 (91.37%) defects, yielding **461/478 (96.44%)** admissible capability-positive mutations and **461/600 (76.83%)** total benchmark yield. Relative to the initially admissible 281/600 (46.83%), this is a $+30.00$ pp absolute increase. Because recovery repair executes the frozen m_0 verbatim and revises only $(w, u, \mathcal{C}_e)$, all 180 rescued candidates retain their original positive forward capability; zero forward-capability regressions were introduced by recovery repair. This fixed-m_0 result does not establish the capability–recoverability trade-off under joint optimization with m.

6 Grounding–Expressivity Factorial Results

Table 2 reports the results of the 2×2 factorial experiment.

6.1 C1: Diagnostic Grounding Unlocks Recoveries on Stratum S_0

On stratum S_0, coarse diagnosis (D_0L_0) yields 0/48 recoveries. Supplying the state-grounded diagnostic bundle (D_1L_0) while keeping the base language L_0 fixed increases successful recovery to **38/48 (79.2%)**. The paired risk difference is $\Delta = +79.17$ pp (95% bootstrap CI $[+66.67, +89.58]$), with exact McNemar $p_{\text{raw}} = 2^{-37} = 7.28 \times 10^{-12}$ ($p_{\text{Holm}} = 1.46 \times 10^{-11}$). For tasks in S_0, resolving the *grounding bottleneck* enables the model to locate and restore the affected state surfaces.

6.2 C2: Language Expressivity Unlocks Recoveries on Stratum S_1

On stratum S_1, which contains tasks recovered by the deterministic L_1 oracle but not by the L_0 oracle, D_0L_0 yields 0/143 recoveries, whereas D_0L_1 recovers **142/143 (99.3%)**. The paired risk difference is $\Delta = +99.30$ pp (95% CI $[+97.90, +100.00]$, $p_{\text{Holm}} = 1.08 \times 10^{-42}$). This demon-

Table 3: Cross-model mechanistic contrasts on the frozen 96-task Qwen3.8-27B replication.

Contrast	Mechanism	Δ (pp)	Holm-adjusted p
C_1: $D_1L_0 - D_0L_0$ on S_0	Diagnostic grounding	$+47.92$	< 0.001
C_2: $D_0L_1 - D_0L_0$ on S_1	Language expressivity	$+37.50$	< 0.001
C_3: $D_1L_1 - D_0L_1$ on S_1	Added grounded diagnostics	$+6.25$	0.375

strates that providing adequate recovery primitives overcomes the *expressivity bottleneck*, allowing the model to synthesize valid recovery programs for nearly all tasks in S_1.

6.3 C3: State-Grounded Diagnostics Interact Non-Monotonically with Language Capacity

Crucially, adding state-grounded feedback on top of the expressive language L_1 does not improve performance. On S_1, recovery drops from 142/143 (99.3%) under D_0L_1 to **133/143 (93.0%)** under D_1L_1. The paired risk difference is $\Delta = -6.29$ pp (95% CI $[-11.19, -2.10]$, exact McNemar $p_{\text{Holm}} = 0.0117$). Trace analysis of the 11 discordant tasks suggests changes in semantic decomposition and operation ordering rather than systematically longer recoveries; the detailed taxonomy and generation metrics appear in Appendix H.4. This analysis is descriptive rather than causal.

6.4 Synthesis Efficiency and Interaction

Synthesis efficiency is detailed in Appendix H.2.

The full-cohort interaction on gpt-oss-120b is $I = (D_1L_1 - D_0L_1) - (D_1L_0 - D_0L_0) = -25.89$ pp (95% CI $[-34.01, -18.27]$): the diagnostic bundle benefits restrictive L_0 but is not additive under L_1.

Cross-model replication. A frozen 96-task Qwen3.8-27B replication with constrained structured decoding preserves the positive grounding and expressivity effects (Table 3; full cells in Appendix H.5).

The negative C_3 interaction observed on gpt-oss-120b did not reproduce under the Qwen3.8-27B constrained-decoding replication configuration ($+6.25$ pp, 95% CI $[-2.08, +16.67]$, $p_{\text{adj}} = 0.375$). Model, decoding protocol, and sample size differ, so the experiment does not isolate the cause.

7 Fresh-Holdout Validation and Robustness

We evaluated frozen L_1 on **300 fresh tasks** (50 per family) with 100 hidden states. Increasing diverse development states from Q_2 to Q_{20} reduced observed false positives from 16.8% to 0.0%, but lowered Rescue@4_{100} from 49.7% to 35.0% ($\Delta = -14.67$ pp, $p = 5.76 \times 10^{-6}$) and increased the conservatism gap from 6.6% to 24.1%.

The strict development gate requires perfect round trips (10/10 in the primary factorial; 20/20 under Q_{20}), potentially rejecting highly reliable candidates. Under Q_{20}, Listeners had 0/50 development passes but 32/50 hidden-criterion passes. This precision–coverage trade-off remains unresolved; zero observed false positives does not guarantee zero population risk. Appendix I gives the rejection-probability illustration and family breakdown.

Additional sweeps over development-set composition, hidden sample size, recovery threshold, and synthesis budget preserve the qualitative precision–coverage trade-off; full results appear in Appendix I.

Fresh-holdout inference cost ranges from 2.39–3.09 calls and 3.37k–4.30k generated tokens per task; detailed latency and per-success costs appear in Appendix I. The final strict-L_0 D_1L_0 run averages 3.51 calls/task.

8 Discussion and Limitations

Under the evaluated medium-reasoning configuration and budget $B = 4$, conventional verifier-guided repair did not overcome the observed representation-bounded failures under L_0 (0/197). No low-, medium-, or high-effort comparison was performed, so the study does not establish the effect of scaling reasoning effort. Moreover, diagnostic feedback interacts non-monotonically with language expressivity on the primary model. These results argue against treating maximal diagnostic specificity as universally preferable; diagnostic granularity should be treated as a configuration-dependent synthesis parameter rather than assumed to be uniformly beneficial. An adaptive diagnostic strategy—escalating from coarse failure indicators to granular state traces only when coarse repair fails—is a plausible mitigation, but no diagnostic curriculum or address-use constraint was evaluated here.

Snapshot and transactional recovery. Effect-scoped snapshots are the stronger baseline in our evaluated serializable in-memory selective-undo regime: they recover 300/300 tasks versus 243/300 for EvoUndo under different-surface changes and 159/300 versus 131/300 under same-surface changes (Appendix D.4). There are no EvoUndo-only wins. EvoUndo targets settings where direct exact effect-scoped restoration is unavailable or insufficient; superiority there is not established. Reported rollback latency measures execution of an already-synthesized recovery and excludes LLM synthesis. No WAL or ARIES runtime was implemented or evaluated.

Limitations and Data Governance: (1) L_0/L_1 are specialized to modeled state types and have no completeness guarantee. (2) Repair fixes $m = m_0$; joint forward/recovery optimization is untested. (3) Distributed and external state, including third-party APIs and unmanaged processes, is unmodeled. (4) Irreversible effects require compensation. (5) Only gpt-oss-120b and Qwen3.8-27B were evaluated. (6) Production witnesses require access controls, encryption, finite retention, least-privilege schemas, and redaction.

9 Conclusion

We introduced EvoUndo, demonstrating that reliable self-evolution requires more than forward capability improvement: verification, state grounding, witness semantics, and recovery-language expressivity must be designed together. When recovery representations align with the underlying state structure, LLMs can reliably synthesize state-dependent recovery programs across counterfactual states, providing a principled foundation for auditable, recoverable self-evolving autonomous agents.

AI Use Statement

Generative AI tools were used for coding assistance and draft editing.

Reproducibility Statement

All code, task definitions, counterfactual generators, protocol locks, and evaluation traces will be released openly. Appendices A–K provide complete architectural schemas, extended related work, language specifications, prompts, baseline comparisons, and statistical derivations.

Ethics Statement

This work develops safety constraints for autonomous agents. Experiments run in isolated sandboxes without human subjects or sensitive production data. Deployment in live infrastructure requires additional authorization and audit controls.

REFERENCES

Emre Can Acikgoz, Cheng Qian, Jonas Hübotter, Heng Ji, Dilek Hakkani-Tür, and Gökhan Tür. Tool-r0: Self-evolving llm agents for tool-learning from zero data. *arXiv preprint arXiv:2602.21320*, 2026.

Debangshu Banerjee, Changming Xu, and Gagandeep Singh. Severa: Verified synthesis of self-evolving agents. *arXiv preprint arXiv:2603.25111*, 2026.

Zhengyu Chen, Teng Xiao, Huaisheng Zhu, Yige Yuan, Luan Zhang, and Jingang Wang. Co-harness: Co-evolving harnesses and model weights for llm agents. *arXiv preprint arXiv:2607.22688*, 2026.

Aaron Councilman, David Jiahao Fu, Aryan Gupta, Chengxiao Wang, David Grove, Yu-Xiong Wang, and Vikram Adve. Towards formal verification of llm-generated code from natural language prompts. *arXiv preprint arXiv:2507.13290*, 2025. URL `https://arxiv.org/abs/2507.13290`.

Herman Errico. Autonomous action runtime management (aarm): A system specification for securing ai-driven actions at runtime. *arXiv preprint arXiv:2602.09433*, 2026.

J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems*, 29(3), 2007. doi: 10.1145/1232420.1232424.

Chris Heunen, Robin Kaarsgaard, and Martti Karvonen. Reversible effects as inverse arrows. *arXiv preprint arXiv:1805.08605*, 2018.

Ying Li, Yanju Chen, Hongbo Wen, Bosi Zhang, Hanzhi Liu, Peiran Wang, Yu Feng, and Yuan Tian. Vigil: Runtime enforcement of behavioral specifications in ai agent skills. *arXiv preprint arXiv:2606.26524*, 2026.

Jiahang Lin, Shichun Liu, Chengjun Pan, Lizhi Lin, Shihan Dou, Xuanjing Huang, Hang Yan, Zhenhua Han, and Tao Gui. Agentic harness engineering: Observability-driven automatic evolution of coding-agent harnesses. *arXiv preprint arXiv:2604.25850*, 2026.

Wenbo Pan, Shujie Liu, Chin-Yew Lin, Jingying Zeng, Xianfeng Tang, Xiangyang Zhou, Yan Lu, and Xiaohua Jia. Evolving agents in the dark: Retrospective harness optimization via self-preference. *arXiv preprint arXiv:2606.05922*, 2026.

Shishir G. Patil, Tianjun Zhang, Vivian Fang, Noppapon C., Roy Huang, Aaron Hao, Martin Casado, Joseph E. Gonzalez, Raluca Ada Popa, and Ion Stoica. Goex: Perspectives and designs towards a runtime for autonomous llm applications. *arXiv preprint arXiv:2404.06921*, 2024.

Juergen Schmidhuber. Gödel machines: Self-referential universal problem solvers making provably optimal self-improvements. *arXiv preprint cs/0309048*, 2003.

Shuai Shao, Kangning Zhang, Qingyao Li, Shijian Wang, Hao Wang, Wenxiang Jiao, Yuan Lu, Yi Guo, Weiwen Liu, and Weinan Zhang. Harness-r1: Learning to edit executable runtime harnesses from agent failure trajectories. *arXiv preprint arXiv:2608.02276*, 2026.

Yifan Shi, Wei Zhang, and Tianyi Cui. A programming paradigm for spatiotemporal composability, 2026. URL `https://github.com/cordiverse/paper`. Preprint.

Haoyu Wang, Christopher M. Poskitt, and Jun Sun. Agentspec: Customizable runtime enforcement for safe and reliable llm agents. *arXiv preprint arXiv:2503.18666*, 2025.

Shuo Yang, Soyeon Caren Han, Xueqi Ma, Yan Li, Mohammad Reza Ghasemi Madani, and Eduard Hovy. Evotool: Self-evolving tool-use policy optimization in llm agents via blame-aware mutation and diversity-aware selection. *arXiv preprint arXiv:2603.04900*, 2026a.

Xianglin Yang, Yufei He, Shuo Ji, Bryan Hooi, and Jin Song Dong. Zombie agents: Persistent control of self-evolving llm agents via self-reinforcing injections. *arXiv preprint arXiv:2602.15654*, 2026b.

Xunjian Yin, Xinyi Wang, Liangming Pan, Xiaojun Wan, and William Yang Wang. Gödel agent: A self-referential agent framework for recursive self-improvement. *arXiv preprint arXiv:2410.04444*, 2024.

Simon Yu, Derek Chong, Ananjan Nandi, Dilara Soylu, Jiuding Sun, Christopher D. Manning, and Weiyan Shi. Shepherd: A runtime substrate empowering meta-agents with a formalized execution trace. *arXiv preprint arXiv:2605.10913*, 2026.

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.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin gödel machine: Open-ended evolution of self-improving agents. In *International Conference on Learning Representations*, volume 2026, pp. 104223–104294, 2026b. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/aa5f5e6eb6f613ec412f1d948dfa21a5-Abstract-Conference.html.

Wangchunshu Zhou, Yixin Ou, Shengwei Ding, Long Li, Jialong Wu, Tiannan Wang, Jiamin Chen, Shuai Wang, Xiaohua Xu, Ningyu Zhang, Huajun Chen, and Yuchen Eleanor Jiang. Symbolic learning enables self-evolving agents. *arXiv preprint arXiv:2406.18532*, 2024.

A Extended Related Work

Self-Improving and Self-Evolving Agents. The concept of systems that iteratively modify their own internal behavior dates back to Schmidhuber's Gödel machine formulation of self-referential improvement (Schmidhuber, 2003). Recent advances in LLMs have made autonomous self-modification empirically practical. Gödel Agent enables an LLM agent to recursively modify its own logic and optimization procedure (Yin et al., 2024), while the Darwin Gödel Machine maintains an evolving population of self-modified coding agents and retains modifications according to downstream task performance (Zhang et al., 2026b). More recent work extends self-improvement from model weights or monolithic agent code to the surrounding executable harness. Self-Harness iteratively mines weaknesses, proposes harness modifications, and validates them using held-out performance (Zhang et al., 2026a), while Agentic Harness Engineering uses explicit editable components, trajectory-derived evidence, and outcome validation to evolve coding-agent harnesses (Lin et al., 2026). Recent work further treats the executable harness itself as an optimization target. Retrospective Harness Optimization (RHO) improves an agent harness from prior trajectories using self-validation and self-preference, without requiring an external labeled validation set (Pan et al., 2026). Co-Harness jointly alternates harness optimization with model-parameter optimization, allowing the runtime scaffolding and the underlying model to co-evolve (Chen et al., 2026). Harness-R1 instead learns a dedicated harness-engineering policy that converts batches of agent failures into validated executable runtime patches optimized for downstream task success (Shao et al., 2026). Related approaches optimize specific parts of the agent system: Agent symbolic learning treats prompts, tools, and pipeline structure as learnable symbolic parameters and updates them using language-based analogues of optimization (Zhou et al., 2024); EvoTool evolves modular tool-use policies through trajectory-grounded failure attribution, targeted mutation, and population selection (Yang et al., 2026a); and Tool-R0 uses self-play to co-evolve task generation and tool-use capability without an initial supervised task dataset (Acikgoz et al., 2026). Across both holistic harness optimization and component-level methods, existing work optimizes forward utility or policy capability; EvoUndo addresses the complementary question of whether persistent model-generated edits retain independently verified recovery semantics.

Reversible Agent Execution and Runtime Effects. Recent agent runtimes have begun treating reversibility as a first-class systems property. GoEX develops a runtime for autonomous LLM actions centered on post-facto validation, providing undo mechanisms and damage confinement so that executed actions can either be reverted or have their possible impact bounded (Patil et al., 2024). Shepherd represents agent–environment interactions as typed effects and supports scoped fork, replay, discard, and rollback of reversible execution state (Yu et al., 2026). Related work on spatiotemporal composability formalizes dynamic software components whose effects can be reverted when components are removed, providing a foundation for dynamically loadable agent harnesses (Shi et al., 2026). These systems provide runtime mechanisms for reversible or compensable effects. EvoUndo studies a distinct challenge that arises when the modification itself is synthesized by an LLM: the required recovery procedure may be unknown, state-dependent, or inexpressible in the available recovery language. We therefore treat witness capture and recovery synthesis as objects of verification rather than assuming that correct inverse semantics are supplied by the component author or runtime.

Reversible Programming and Bidirectional State Transformation. Reversible computation and bidirectional programming study transformations for which information required to reconstruct prior states is preserved or explicitly represented. Reversible-effect systems extend these ideas to stateful computations and mutable effects (Heunen et al., 2018). Classical bidirectional transformations, particularly lenses, formalize paired transformations that propagate updates between related state representations while satisfying consistency laws (Foster et al., 2007). EvoUndo shares the need to retain sufficient prior-state information for reverse state transformation, but differs because its forward edits and recovery programs are model-generated, and recovery correctness is evaluated empirically across counterfactual harness states. Consequently, a non-injective forward mutation need not be rejected when a state-dependent witness preserves the information required for recovery.

Verification of Agent Modifications. Self-improving agents commonly validate candidate modifications through downstream benchmarks, regression tests, or deterministic evaluation, including population-based self-modification (Zhang et al., 2026b) and harness optimization (Zhang et al., 2026a; Lin et al., 2026). Such validation can determine whether a modification improves capability without establishing whether its persistent effects can be removed. EvoUndo separates these properties: forward capability is evaluated through the task objective, while recovery is independently evaluated through counterfactual round trips; development states provide diagnostics, while hidden-state contents and per-state outcomes are withheld. In the primary factorial, aggregate hidden IID/OOD LCB summaries are nevertheless returned during repair. This distinction is central: a mutation may improve task performance and pass ordinary regression tests while still destroying information, leaking state, or requiring recovery operations unavailable in the current representation.

Formal Verification of Self-Evolving Agents. SEVerA studies self-evolving agent synthesis under formal correctness constraints, using formally guarded generative-model calls and program verification to ensure that synthesized agents satisfy specified contracts independently of learned model parameters (Banerjee et al., 2026). This provides stronger deductive guarantees than EvoUndo's empirical counterfactual verification. The two approaches address complementary properties: SEVerA verifies that generated agent programs satisfy formal behavioral constraints, whereas EvoUndo asks whether persistent, model-generated harness mutations can be returned to their prior observational state using state-dependent witness and recovery semantics.

Runtime Safety Enforcement. AARM treats agent action execution as a security boundary and proposes a runtime control layer that intercepts actions before execution, evaluates them against authorization and intent policies, and records execution evidence (Errico, 2026). AgentSpec similarly introduces a lightweight domain-specific language for specifying and enforcing runtime safety constraints on LLM agents through explicit triggers, predicates, and enforcement actions (Wang et al., 2025). VIGIL similarly converts behavioral specifications into executable runtime policies over agent–tool traces, including temporal, argument, and value-flow constraints (Li et al., 2026). Such systems determine whether an action sequence satisfies a behavioral policy; EvoUndo targets a different temporal property: whether a persistent harness self-modification can later be removed while restoring the relevant prior harness state. Runtime policy enforcement and recoverability-constrained evolution can therefore be composed. Persistent self-evolution also creates security risks beyond per-action authorization: Zombie Agents demonstrates that attacker-controlled content can be incorporated into an agent's evolving long-term memory and persist across sessions, later inducing unauthorized behavior (Yang et al., 2026b). EvoUndo studies whether persistent harness modifications carry recovery semantics that permit their effects to be independently verified and removed.

Proof-based verification and runtime enforcement. Astrogator verifies LLM-generated Ansible programs against a formal query using symbolic interpretation, providing deductive guarantees within its specified automation calculus (Councilman et al., 2025). EvoUndo instead provides empirical typed round-trip evidence over heterogeneous modeled harness states; this covers recovery properties without claiming proof-level completeness. AgentSpec enforces per-action runtime policies through triggers, predicates, and enforcement actions (Wang et al., 2025); such governance is orthogonal to whether an admitted persistent mutation can later restore prior state, so the two mechanisms can be composed.

B EvoUndo Implementation Details

B.1 Harness State Representation

We represent the agent harness as a typed persistent state containing configuration, prompt templates, routing state, tool registries, middleware sequences, event-listener bindings, sandboxed files, and managed resources. Each state target is associated with a canonical addressing scheme and a type-specific observational equivalence policy. The global frame condition in Equation 1 additionally checks all modeled persistent state outside the effect contract.

B.2 Candidate Representation

Each EvoUndo candidate is represented as $\xi = (m, w, u, \mathcal{C}_e)$, where m is the frozen forward mutation, w is the witness-capture program, u is the recovery program, and $\mathcal{C}_e$ is the effect contract. In EvoUndo recovery repair, the forward mutation m is strictly immutable ($m_{\text{repaired}} \equiv m_0$). Only the witness capture program w, recovery program u, and declared effect contract $\mathcal{C}_e$ may be revised across repair turns. The contract is constrained to the runtime's typed effect-contract vocabulary; when omitted, it is inferred from the locked forward operations. The runtime independently computes $E(m, s) = \text{SnapshotDiff}(s, m(s)) \cup \text{ExecutionTraceEffects}(m, s)$ and requires $E(m, s) \subseteq \mathcal{C}_e$: an omitted observed effect causes rejection, while over-declaration expands the state scope that must satisfy recovery equivalence.

Witness and resource execution. The capture program records the prior value and a boolean `{key}_existed`. A recovery primitive dynamically restores the old binding when this flag is true and deletes/unregisters the mutation-introduced binding when it is false; this is not an unconditional inverse. For managed resources, $w(s)$ stores lightweight prior-allocation descriptors, while forward execution appends allocation receipts to `created_resources`. Recovery consumes both components and resolves symbolic resource keys through the runtime registry before explicit handle closure and unbinding. No future handle is attributed to the pre-state witness.

B.3 Canonical State Addresses

Verifier diagnostics use canonical semantic state locations rather than raw implementation-specific object identities. Representative addresses include `tools["tool_id"]`, `config["key"]`, `middleware["middleware_id"]`, `files["path"]`, and `resources["resource_id"]`. Canonicalization is applied uniformly across verifier residual extraction and diagnostic rendering. Diagnostic addresses exposed to the recovery model are derived only from verifier-visible development executions.

C Recovery Languages

C.1 Base Recovery Language L_0

The original recovery language supports local operations for configuration, prompt, tool-registry, and routing state. Its implemented capture, forward, and recovery primitives are enumerated in Table 1. The L_0 runtime explicitly rejects operations requiring middleware-sequence, listener, file-prestate, socket-descriptor, or ordered multi-surface semantics.

C.2 Extended Recovery Language L_1

L_1 strictly extends L_0 with five implemented capability classes identified by the natural-failure solvability audit:

1. **Indexed sequence capture and restoration**: preserves both sequence membership and ordering information.
2. **Listener capture and restoration**: records the prior event binding and restores it with `OpSpec(op_type="restore_listener", target="<event_name>", witness_key="<w_key>")` or removes a newly registered listener with `OpSpec(op_type="unregister_listener", target="<event_name>")`.
3. **File pre-state capture and restoration**: captures whether a file existed and, when necessary, its previous contents or state.
4. **Resource-descriptor capture and restoration**: records prior binding existence and lightweight allocation descriptors; recovery also consumes the dynamic receipt produced by forward execution. Live sockets are not serialized into the pre-state witness.
5. **Ordered multi-surface recovery**: permits recovery of interacting state mutations in an explicitly specified order.

The runtime enforces the requested language level. An L_1-only primitive emitted in an L_0 condition is classified as an invalid model output and consumes the corresponding synthesis attempt. These are implemented operational semantics and do not establish formal language completeness.

D Counterfactual Verification and Audit Baselines

D.1 Round-Trip Evaluation and Counterfactual Generation Taxonomy

For every counterfactual state s, EvoUndo performs $w_{\text{pre}} = w(s)$, $(s', \mathcal{R}_m(s)) = [\![m]\!](s)$, and $\hat{s} = u(s', w_{\text{pre}}, \mathcal{R}_m(s))$, and tests $\hat{s} \simeq_{\mathcal{C}_e} s$. Development states produce state-level verifier feedback. In the original factorial, hidden state contents and per-state outcomes remain withheld, but aggregate IID/OOD LCB summaries are visible during repair. The fresh-holdout experiment uses a fully feedback-isolated hidden set.

Table 4 details the exact generation strategies implemented in `CounterfactualStateGenerator` across development, IID hidden, and OOD hidden splits.

Table 4: Taxonomy of counterfactual state perturbation strategies across modeled state targets.

Regime	Perturbation Strategy	Operational Realization
IID	Boundary Extreme States	Keys deleted, extreme scalars ($T \in \{0.0, 1.0\}$, timeouts $\in \{0, 300\}$, empty/large files)
IID	Prior-Existence Branches	Tools absent vs. pre-existing vs. conflicting signatures; listeners empty vs. populated
IID	Combinatorial Assignments	Joint multi-key randomized configuration, tool registry, and file-state permutations
IID	Dependency Sensitivities	Model routing fallbacks (`default_model`, `fallback_model` bindings)
IID	Adversarial Noise States	Uncontracted irrelevant config keys, uncontracted file creation, permission flags
OOD	Unfamiliar Nested Keys	Injecting unseen hierarchical configuration namespaces (`custom_subsystem_i.nested.enabled`)
OOD	Conflicting Socket Bindings	Pre-allocating external port handles (`tcp://127.0.0.1:91xx`) to induce resource contention
OOD	High-Priority Middleware	Prepending unmodeled high-priority interceptors (`priority=1`) into the middleware chain
OOD	Deep Directory Structures	Creating multi-level nested directory hierarchies (`external/cache/partition_i/meta.json`)
OOD	Inverted Routing Policies	Reversing default model dispatch and explicitly disabling fallback routes

D.2 Typed Observational Equivalence

Recovery follows Equation 1: type-specific restoration rules apply within the declared effect contract $\mathcal{C}_e$, and all modeled persistent state outside it must retain strict canonical equality with the pre-state.

For configuration entries, equivalence requires restoration of the relevant key and value state. Tool and registry-like state requires restoration of the relevant binding, including the registered identity and associated implementation state represented by the harness. Middleware and other sequence-like state requires both membership and ordering to match the pre-mutation state. Listener state requires restoration of the relevant registration state. File-backed effects require restoration of the relevant pre-state, including existence and file state when represented by the recovery contract. Resource-backed effects require restoration of the relevant descriptor and lifecycle state represented by the harness. For compound mutations touching multiple state targets, every contract-relevant target must satisfy its corresponding equivalence policy. The verifier compares the modeled persistent harness state, using typed equivalence inside the contract and strict canonical equality outside it. Final-state checks do not independently constrain transient recovery-side effects that disappear before verification.

D.3 Effect-Contract Coverage Audit

We conducted a controlled audit to evaluate whether dynamic effect tracking reliably detects undeclared side effects. The runtime constructs $E(m, s) = \text{SnapshotDiff}(s, m(s)) \cup \text{ExecutionTraceEffects}(m, s)$ and requires $E(m, s) \subseteq \mathcal{C}_e$; any observed effect omitted from the declared contract therefore causes rejection. For 300 mutations spanning all six harness families, we constructed intentionally incomplete effect contracts by omitting at least one ground-truth affected target E_{true}. Dynamic effect tracking detected every injected undeclared semantic effect (300/300, 100.0%), with zero false positives across 300 matched complete contracts (0/300, 0.0%). In an additional 300 nondeterministic runtime controls (varying OS thread IDs, timestamps, and memory pointer representations), the canonicalizer produced zero false alarms. These guarantees apply to the

modeled harness state space and do not extend to unmodeled external OS processes or out-of-band network state.

D.4 Snapshot-Restore Baselines

To compare EvoUndo against direct runtime snapshotting, we used the frozen 300-task fresh cohort (50 tasks per benchmark family, no overlap with the original 600 tasks) in two serially interleaved protocols. First, mutation m_1 changes the harness; a later legitimate mutation m_2 then changes state. Recovery attempts to undo only m_1, while valid later effects of m_2 must survive:

$$s_{\text{pre}} \xrightarrow{m_1} s_{\text{fwd}} \xrightarrow{m_2} s_{\text{later}} \xrightarrow{u_1} s_{\text{rec}}.$$

Success requires restoration of m_1's effects, preservation of m_2's effects, and no unintended changes elsewhere. In the different-surface protocol, the intervention generator selects a surface disjoint from m_1's declared and observed effects. In the same-surface protocol, it changes a different key or element within the same affected surface/container (for example, appending another middleware item). Concurrent here denotes intervening changes, not simultaneous execution. A full-state rollback loses the later valid changes and therefore fails selective undo, even if it exactly restores the old snapshot.

Table 5: Comparison of EvoUndo against snapshot-restore baselines across 300 tasks.

Method	Different-Surface Selective	Same-Surface Concurrent	Mean Storage	Runtime rollback latency (ms)
Full Snapshot	0/300 (0.0%)	0/300 (0.0%)	1165.0 B	0.084 ms
Effect-Scoped Snapshot	**300/300 (100.0%)**	**159/300 (53.0%)**	**177.8 B**	**0.039 ms**
EvoUndo (Synthesized Recovery)	243/300 (81.0%)	131/300 (43.7%)	393.6 B	0.050 ms

Runtime rollback execution for an already-synthesized recovery program; excludes offline LLM recovery-program synthesis.

Storage is measured as UTF-8 encoded JSON bytes representing the entire harness state for Full Snapshot, touched pre-state values for Effect-Scoped Snapshot, and the witness store plus recovery AST for EvoUndo. Corresponding mean/median storage is 1165.0/1165.0 B, 177.8/145.0 B, and 393.6/350.5 B, respectively. Table 5 reports mean runtime rollback execution latency, measured with `time.perf_counter()`, for an already-synthesized recovery program. The means are 0.084 ms, 0.039 ms, and 0.050 ms, respectively. Offline LLM synthesis is excluded; model-call costs are reported separately in Section 7.

Full snapshots are too coarse: restoring a full snapshot clobbers all subsequent state changes (0/300 success). Effect-scoped snapshots are the strongest option here when the exact affected pre-state is known, serializable, and directly restorable. There were zero EvoUndo-only wins and 57/28 snapshot-only wins in the different-/same-surface comparisons. EvoUndo targets settings where direct effect-scoped restoration is unavailable or insufficient, including structured, state-dependent, or non-serializable inverse semantics; this benchmark does not demonstrate superiority in those settings. Both selective approaches degrade when subsequent mutations occur within the same structured surface. No WAL or ARIES runtime was implemented or evaluated; transactional logging is discussed only as conceptual related work.

E Controlled Recoverability Benchmark

The controlled benchmark contains 120 deliberately recovery-defective but capability-positive candidates. The forward mutation is held fixed and the defective recovery specification is produced by corrupting witness or recovery semantics.

The benchmark includes missing witness capture, dummy witness capture, incorrect recovery operations, resource leakage, and hidden-effect omission. Every benchmark item has a known recoverable counterpart under the available recovery representation, allowing the experiment to isolate recovery synthesis from language expressivity.

At budget $B = 4$, the principal recovery results are:

- Strict regeneration: 4/120 (3.3%)

- Generic feedback: 104/120 (86.7%)
- Raw verifier traces: 101/120 (84.2%)
- Typed diagnosis: 114/120 (95.0%)
- Prescriptive guidance: 117/120 (97.5%)

F Natural Failure Bank

Turn-0 generation used temperature 0.2, top-p 0.95, a 2,048-token limit, seed 7000+ task index, and structured Outlines JSON generation.

We construct the natural-failure population by single-sample zero-shot generation (zero demonstrations, one candidate per task, no rerolls) for 600 unseen self-evolution tasks, using the six 100-task benchmark families defined in Section 4. The resulting population is:

- 281/600 already admissible,
- 197/600 capability-positive but recovery-defective,
- 122/600 capability inadequate.

The 197 recovery-defective candidates are frozen before subsequent repair, solvability, and factorial analyses. Multi-Surface is a composite benchmark family, not an atomic harness surface; Config contributes zero recovery-defective cases. The family/stratum counts appear in Table 6. Under the original recovery representation, Generic, Raw, Typed, and Prescriptive repair each recover 0/197 natural failures at budget $B = 4$. Strict regeneration recovers 6/197.

Table 6: Frozen primary cohort by benchmark family. Each family contains 100 tasks. Adm. denotes initially admissible, Inad. capability-inadequate, Def. recovery-defective, and Resid. unrecovered by the implemented L_1 oracle.

Family	Adm.	Inad.	Def.	S_0	S_1	Resid.
Config	96	4	0	0	0	0
Tools	38	15	47	45	0	2
Middleware	1	24	75	0	72	3
Listeners	91	0	9	0	9	0
Resources	26	39	35	0	35	0
Multi-Surface	29	40	31	3	27	1
Total	281	122	197	48	143	6

G Oracle Solvability Audit Methodology and Robustness

To distinguish synthesis search failures from empirical representational limitations under the implemented recovery languages, we conducted a zero-generation constructive oracle solvability audit across all 197 frozen natural recovery failures.

G.1 Deterministic Constructive Principles

The solvability oracle is fully deterministic and non-generative. Specifically:

- **No LLM calls**: The oracle contains no stochastic generation or learned model inference.
- **No search or enumeration**: Recovery programs are constructed directly via deterministic structural compilation rather than heuristic search, beam search, or candidate enumeration.
- **No retries or manual task tuning**: Every candidate undergoes a single-pass deterministic transformation using fixed AST construction heuristics.

Disclaimer: This is empirical recoverability under the implemented deterministic constructive oracle, not a formal proof of language-theoretic completeness.

G.2 Oracle Inputs and Construction Algorithm

The constructive oracle uses the following information:

1. The frozen forward mutation abstract syntax tree (AST) m.
2. The pre-mutation harness state s.
3. Forward-operation surfaces and observed mutation effects, from which it constructs its own effect contract.
4. The evaluator state distributions $\mathcal{Q}_{\text{dev}}$ ($|\mathcal{Q}_{\text{dev}}| = 10$) and $\mathcal{Q}_{\text{hid}}$ ($|\mathcal{Q}_{\text{hid}}| = 40$).

The oracle does not blindly inherit the candidate's declaration. It constructs

$$\mathcal{C}_e^{\text{oracle}} = \left(\bigcup_{\text{op} \in m} \text{Surface}(\text{op}) \right) \cup E(m, s_\star).$$

This tight scope is derived from forward operation surfaces and observed effects. Its construction remains subject to the implemented AST heuristics described below.

For each forward operation $f_i \in m$:

1. **Witness Construction**: The oracle deterministically constructs a witness capture operation w_i that queries and stores the pre-mutation state of the targeted surface address.
2. **Inverse Construction**: The oracle deterministically constructs the corresponding inverse recovery primitive u_i permitted under the target language grammar $\mathcal{L}$.
3. **LIFO Unwinding Execution**: The synthesized recovery program orders inverse operations in exact reverse (Last-In, First-Out) order: $u = (u_k, \ldots, u_1)$.

G.3 Language Expressivity Boundaries

Base Recovery Language L_0. Supports the implemented configuration, prompt, tool-registry, and routing primitives enumerated in Table 1. It strictly prohibits middleware-sequence, listener, file-prestate, socket-descriptor, and ordered multi-surface operations.

Extended Recovery Language L_1. Extends L_0 with:

1. **Indexed Sequence Restoration**: Preserves sequence membership, element identity, and exact insertion index for ordered lists (e.g., middleware chains).
2. **Listener Capture and Restoration**: Uses `capture_listener` with `restore_listener` or `unregister_listener` for the distinct event-listener surface.
3. **File Pre-State Preservation**: Captures pre-existence status and SHA-256 validated content buffers.
4. **Resource Descriptor Management**: Captures lightweight prior-allocation descriptors; mutation-produced receipts bind symbolic resource keys to handles for explicit closure and unbinding.
5. **Ordered Multi-Surface Unwinding**: Executes the implemented recovery operations in explicit reverse-dependency order.

G.4 Admission Criterion for Oracle Solvability

A mutation m is classified as empirically recoverable under language $\mathcal{L}$ if and only if:

1. The synthesized (w, u) candidate compiles strictly within the language gate for $\mathcal{L}$.
2. Forward capability is preserved: $\Delta J(m; s_\star) > 0$.
3. Round-trip verification achieves 10/10 (100.0%) on development states $\mathcal{Q}_{\text{dev}}$.

4. Each of the separate 20-state hidden IID and OOD splits satisfies the executed Wald lower-endpoint threshold $\tau_R = 0.85$, equivalently at least 19/20 successes on each (Appendix J.1).

5. Recovered states pass identical typed observational equivalence $\hat{s} \simeq_{\mathcal{C}_e} s$.

Six residual oracle failures. The L_1 constructive oracle recovered 191/197 tasks. The remaining six comprise four middleware index-targeting cases (three Middleware-family tasks and one Multi-Surface task) and two tool registry-key versus ID naming mismatches in the AST heuristic. These are implemented-oracle failures, not a completeness result or a claim that the tasks are theoretically unrecoverable (Table 7).

Table 7: Six tasks unrecovered by the implemented L_1 oracle. IDs use task-ID prefixes. These are constructive-heuristic failures, not theoretical impossibility results.

Task prefix	Benchmark family	Heuristic failure class
`nat_mw_003`	Middleware	Middleware index targeting
`nat_mw_055`	Middleware	Middleware index targeting
`nat_mw_076`	Middleware	Middleware index targeting
`nat_multi_041`	Multi-Surface	Middleware index targeting
`nat_tool_070`	Tools	Registry key versus tool ID
`nat_tool_073`	Tools	Registry key versus tool ID

G.5 Representative Case Studies

Case A: Configuration illustration (outside the primary failure strata). Configuration contributed zero recovery-defective cases to the frozen primary cohort, so this scalar restoration example is not a member of S_0. The illustrated forward mutation executes `set_config("max_retries", 5)`. The deterministic constructive oracle records the pre-mutation scalar value with `capture_config` and constructs the corresponding `restore_config` recovery operation using that witness. This candidate compiles and satisfies full round-trip equivalence under both L_0 and L_1.

Case B: Tool Registration in S_0 (`nat_tool_004_003`). The forward mutation registers a new diagnostic tool `tool_003_dependency_grapher` (`nat_tool_004_003_dependency_grapher`). The oracle records the pre-mutation registry state with `capture_tool` and constructs recovery that restores the prior binding if it existed, or uses `unregister_tool("tool_003_dependency_grapher")` when it was newly introduced. Because single-tool unregistration is supported in the base grammar, the recovery compiles and succeeds under L_0.

Case C: Middleware Sequence in S_1 (`nat_mw_001_000`). The forward mutation inserts a sliding-window rate limiter into the middleware pipeline (`nat_mw_001_000_rate_limiter_sliding_window`). Reversing this sequence mutation requires the indexed `capture_middleware` and `restore_middleware` operations. Under L_0, these operations are outside the grammar and rejected by the language gate. Under L_1, the oracle constructs the valid sequence restoration program, achieving 100% recovery on hidden counterfactuals.

Case D: Multi-Surface Task in S_1 (`nat_multi_036_mw_and_event`). The forward mutation combines an event callback registration with middleware insertion; recovery follows the reverse order of the actual forward operations. Under L_0, recovery fails because multi-surface coordination is outside the L_0 grammar. Under L_1, the oracle constructs an ordered LIFO unwind program using listener restoration/unregistration and middleware restoration in reverse forward-operation order, satisfying the observational contract $\mathcal{C}_e$ across all 40 hidden states.

G.6 Oracle Stratum Resampling Stability Sweep

To evaluate whether the empirical strata S_0 and S_1 are sensitive to the specific counterfactual pre-states sampled in $\mathcal{Q}_{\text{dev}}$ and $\mathcal{Q}_{\text{hid}}$, we re-evaluated the deterministic constructive oracle across 10 independently resampled counterfactual seeds (seeds 1000 through 10000) using the same generative state distribution. A distinct counterfactual suite was generated for each seed. Across all 10 independent resamplings, stratum membership was unchanged ($|S_0| = 48$, $|S_1| = 143$, unrecovered $= 6$, Jaccard $= 1.0000$ for both S_0 and S_1, with zero stratum migrations across 1,970 task-seed evaluations). This establishes empirical stability under independent resampling from the implemented generator; it does not establish completeness, distribution-independent task labels, or invariance under arbitrary OOD, adversarial, or structurally different counterfactual generators.

H Factorial Protocol and Additional Results

H.1 Diagnostic Prompt and Model-Call Configuration

The system prompt is identical in all four factorial cells:

```
You are an expert self-evolving agent compiler generating
verified repairs.
```

Under D_0, each repair call receives the following diagnostic fields:

```
Verification Result: RECOVERY_EQUIVALENCE_FAILED
Affected Subsystems: <task.surfaces>
Defect Summary: The candidate executed forward mutations on
<task.surfaces> without registering complete, sound inverse
capture and recovery operations.
Dev Recovery Rate: <dev_recovery_rate * 100>% (Required: 100.0%)
Hidden IID LCB: <hidden_iid_lcb> (Required: >= 0.85)
Hidden OOD LCB: <hidden_ood_lcb> (Required: >= 0.85)
```

The D_1 condition uses the same diagnostic content and additionally appends the composite state-grounded diagnostic bundle below. The literal protocol identifier is preserved:

```
Diagnostic Mode: D1_EXACT_ADDRESS
Forward Effect Order: <forward_effect_order>
Observed State Residuals: <observed_state_residuals>
Residual Count: <residual_count>
Verifier Observation: <diagnostic_evidence>
```

All four factorial cells use the same model-call configuration: `openai/gpt-oss-120b` at the frozen experimental revision, temperature 0.2, medium reasoning effort, and an explicit maximum of 2,048 generated tokens per call. The model's native 131,072-token context window is available without additional input truncation.

H.2 Synthesis Efficiency and Budgeted Recovery Progression

Synthesis efficiency $\eta(D, L) = \frac{\text{model rescues}}{\text{oracle recoverable}}$ separates failures of model synthesis from failures of the recovery representation. The denominator counts only tasks for which the deterministic oracle establishes that the assigned language can express an admissible recovery. Efficiency is therefore 0.0% for D_0L_0, rises to 79.2% for D_1L_0, and reaches **94.2%** for D_0L_1. Adding state-grounded diagnostics to the rich language lowers efficiency to 87.4%. Thus, once representational feasibility is held fixed, coarse diagnosis with the expressive language realizes the largest fraction of available recoveries.

For the final strict-L_0 D_1L_0 execution, cumulative rescue progresses as follows: 24/197 at $B = 1$, 35/197 at $B = 2$, 37/197 at $B = 3$, and 38/197 at $B = 4$. All 38 successes belong to S_0; no S_1 task or task unrecovered by the L_1 oracle passes under strict L_0.

H.3 LANGUAGE-GATE BEHAVIOR

Recovery-language assignments are enforced before execution. For each generated recovery program, the runtime gate checks every proposed primitive against the operation set authorized by the assigned language. Under an L_0 condition, primitives requiring indexed sequence restoration, listener restoration or unregistration, file pre-state restoration, socket-descriptor restoration, or ordered multi-surface recovery are L_1-only and are therefore rejected rather than executed. An out-of-language proposal consumes the corresponding synthesis attempt.

Across 691 recorded model responses in D_1L_0, 290 attempted at least one L_1-only recovery primitive and were rejected by the language gate: 21/144 (14.6%) on S_0 versus 253/523 (48.4%) on S_1. The remaining 16 rejections occurred on the six tasks outside the oracle-defined S_0/S_1 strata.

H.4 DISCORDANT TRACE ANALYSIS (D_1L_1 VS. D_0L_1)

We analyzed the 11 discordant tasks between D_0L_1 and D_1L_1 on stratum S_1 (10 D_0L_1-only successes and 1 D_1L_1-only success). Table 8 reports the failure taxonomy alongside quantitative trace metrics extracted from execution logs.

Table 8: Structural failure taxonomy of the 11 discordant tasks on S_1. Operations are emitted patch operations for D_1L_1 and compiled recovery AST operations for D_0L_1. The aggregate covers the ten D_0L_1-only successes.

			Mean operations		Completion tokens	
Category	Tasks	Winner	D_1L_1	D_0L_1	D_1L_1	D_0L_1
Over-decomposition	5	D_0L_1	1.80	2.40	739.0	625.2
Sequence inversion	3	D_0L_1	2.00	4.00	580.3	828.3
Over-capture clobbering	2	D_0L_1	2.00	2.00	414.5	759.5
Ambiguous target	1	D_1L_1	4.00	2.00	707.0	751.0
Aggregate (D_0L_1 wins)	10	D_0L_1	1.90	2.80	626.5	713.0

H.5 CROSS-MODEL REPLICATION WITH QWEN3.8-27B

To evaluate whether the observed grounding and expressivity mechanisms generalize across model architectures, we conducted an independent cross-model replication on `Qwen3.8-27B` under constrained structured decoding, using the original `gpt-oss-120b` experiment as historical reference. The replication was conducted on a frozen 96-task subset (48 tasks from S_0 and 48 tasks from S_1). All replication runs enforced constrained CandidateAST JSON decoding with a hard cap of 2,048 generated tokens per call under identical prompt schemas. Only the Qwen model was executed in this replication; `gpt-oss-120b` metrics on the matching 96-task subset are provided for historical reference. An earlier pilot run of Qwen with unconstrained decoding exhibited syntax drift and is excluded from confirmatory analysis.

Table 9: Cross-model factorial replication on the 96-task frozen subset using Qwen3.8-27B with constrained structured decoding. Historical gpt-oss-120b numbers on the same 96 tasks are shown for comparison.

Factorial Cell	**Qwen3.8-27B** (S_0)	**Qwen3.8-27B** (S_1)	**gpt-oss-120b** (S_0)	**gpt-oss-120b** (S_1)
D_0L_0	3/48 (6.2%)	0/48 (0.0%)	0/48 (0.0%)	0/48 (0.0%)
D_1L_0	26/48 (54.2%)	0/48 (0.0%)	38/48 (79.2%)	0/48 (0.0%)
D_0L_1	35/48 (72.9%)	18/48 (37.5%)	38/48 (79.2%)	47/48 (97.9%)
D_1L_1	43/48 (89.6%)	21/48 (43.8%)	34/48 (70.8%)	46/48 (95.8%)
Paired Mechanistic Contrasts (Qwen3.8-27B)				
C_1 (D_1L_0 vs. D_0L_0 on S_0)	$\Delta = +47.92$ pp (95% CI $[+31.25, +64.58]$), exact McNemar $p_{\text{adj}} < 0.001$			
C_2 (D_0L_1 vs. D_0L_0 on S_1)	$\Delta = +37.50$ pp (95% CI $[+25.00, +52.08]$), exact McNemar $p_{\text{adj}} < 0.001$			
C_3 (D_1L_1 vs. D_0L_1 on S_1)	$\Delta = +6.25$ pp (95% CI $[-2.08, +16.67]$), exact McNemar $p_{\text{adj}} = 0.375$			

In the D_1L_0 condition on Qwen3.8-27B, a 10.2% truncation rate was observed when detailed address diagnostics combined with complex repair logic approached the 2,048-token generation limit. Despite lower absolute yield compared to gpt-oss-120b, C_1 and C_2 replicate across the evaluated backbones, providing cross-model evidence that state grounding and recovery-language expressivity are recurring bottlenecks. The negative D_1-on-L_1 interaction observed on gpt-oss-120b does not replicate on Qwen3.8-27B ($\Delta = +6.25$ pp, 95% CI $[-2.08, +16.67]$, $p_{\text{adj}} = 0.375$) under that constrained-decoding replication configuration. Model, decoding protocol, and sample size differ from the primary study; the comparison does not isolate the cause of the difference.

I Fresh-Holdout Counterfactual Sensitivity

We evaluated the frozen L_1 recovery language on 300 fresh tasks, with 50 tasks from each of six architectural families and zero overlap with the original 600-task cohort. All conditions used the same frozen forward mutation, D_0 diagnostics, recovery language, model configuration, repair budget $B = 4$, and common 100-state hidden evaluation set.

Development-gate conservatism. The development gate requires perfect round trips (10/10 in the primary factorial and 20/20 under Q_{20}), while the hidden criterion tolerates some failures. Under an illustrative independent-trial model, true per-state reliability $p = 0.96$ gives rejection probability $1 - p^{20} \approx 55.8\%$, versus $1 - p^2 \approx 7.8\%$ for Q_2. This explains why high-reliability candidates may fail the strict development gate; these probabilities illustrate the policy rather than estimate a candidate's true reliability. Under Q_{20}, Listeners had 0/50 development passes but 32/50 hidden-criterion passes; Resources contributed the other 15 of the 47 conservatively rejected tasks. Zero observed false positives in that condition does not guarantee zero population risk. This precision–coverage trade-off remains unresolved.

Sensitivity and Robustness Analysis. Robustness of the recovery framework was comprehensively evaluated across multiple operational dimensions: synthesis budgets $B \in \{1, 2, 3, 4\}$ (Appendix H), recovery admission thresholds $\tau_R \in \{0.75, 0.80, 0.85, 0.90, 0.95\}$ (Table 12), development set configurations (Q_2, Q_5, Q_{10}, Q_{20}, and Q_{10}-Narrow, Table 10), and hidden sample sizes $n_{\text{hid}} \in \{10, 20, 30, 50, 100\}$ (Table 11). Across these sweeps, the qualitative behavior of counterfactual verification remains stable: stricter development verification improves precision at the cost of synthesis yield, while admission remains reasonably stable under moderate threshold choices. The grounding and expressivity effects are independently supported by the protocol-locked factorial and cross-model replication. Increasing the size of the diverse development set ($Q_2 \to Q_{20}$) eliminated observed false positives in this study ($16.8\% \to 0.0\%$) at the cost of expanding the conservatism gap ($6.6\% \to 24.1\%$). Recovery yields remain broadly stable across moderate thresholds $\tau_R \in [0.75, 0.90]$ (168 to 142 rescues under Q_2), but setting a stringent threshold of $\tau_R = 0.95$ materially depresses end-to-end yield (dropping to 102 rescues), indicating that recovery yield is threshold-sensitive at extreme stringencies rather than strictly invariant.

Inference and verification overhead. Across the 300-task fresh holdout, inference cost ranges from 2.39–3.09 model calls and approximately 3.37k–4.30k generated tokens per task, with higher verification strictness increasing cost (Table 13 in Appendix I). On the primary 197-task factorial cohort, closed-loop repair averaged 1.78 calls/task in D_0L_1 (351/197), 1.95 in D_1L_1 (385/197), and 3.51 in the final strict-L_0 D_1L_0 rerun (691/197), with mean model call latencies between 5.25s and 5.85s on H200 GPUs. Detailed fresh-holdout token counts, median latencies, and tokens per successful rescue are reported in Appendix I.

J Statistical Analysis

J.1 Executed Admission Rule and Post-Hoc Rescoring

The primary 197-task protocol independently evaluates 20 hidden IID and 20 hidden OOD states. For each split,

$$\text{LCB}_{\text{Wald}}(k, n) = \max\left(0, \hat{p} - 1.96\sqrt{\frac{\hat{p}(1-\hat{p})}{n}}\right), \qquad \hat{p} = k/n.$$

Table 10: Fresh-holdout counterfactual sensitivity on 300 unseen tasks. False positives are development-passing candidates that fail the 100-state hidden admission criterion. The conservatism gap counts candidates rejected by development verification that nevertheless satisfy the hidden criterion post hoc.

Condition	Dev Pass	False Positive	Rescue@4_{100}	Hidden Criterion	Conservatism Gap
Q_2-Diverse	179/300 (59.7%)	30/179 (16.8%)	149/300 (49.7%)	157/300 (52.3%)	8/121 (6.6%)
Q_5-Diverse	168/300 (56.0%)	17/168 (10.1%)	151/300 (50.3%)	165/300 (55.0%)	14/132 (10.6%)
Q_{10}-Diverse	134/300 (44.7%)	8/134 (6.0%)	126/300 (42.0%)	152/300 (50.7%)	26/166 (15.7%)
Q_{20}-Diverse	105/300 (35.0%)	0/105 (0.0%)	105/300 (35.0%)	152/300 (50.7%)	47/195 (24.1%)
Q_{10}-Narrow	173/300 (57.7%)	20/173 (11.6%)	153/300 (51.0%)	153/300 (51.0%)	0/127 (0.0%)

Table 11: Hidden counterfactual sample-size sensitivity. Each entry is mean empirical recovery rate / mean 95% Wilson lower confidence bound across 300 fresh tasks. Prefixes are nested subsets of the same 100-state hidden pool.

Condition	$n_{\text{hid}} = 10$	$n_{\text{hid}} = 20$	$n_{\text{hid}} = 30$	$n_{\text{hid}} = 50$	$n_{\text{hid}} = 100$
Q_2-Diverse	60.5 / 42.2	60.6 / 49.3	60.7 / 52.3	60.7 / 55.1	60.7 / 57.5
Q_5-Diverse	60.8 / 42.8	61.0 / 50.1	61.0 / 53.1	61.2 / 55.9	61.2 / 58.2
Q_{10}-Diverse	56.8 / 40.0	57.1 / 46.8	57.1 / 49.6	57.2 / 52.2	57.3 / 54.4
Q_{20}-Diverse	56.6 / 39.8	56.7 / 46.4	56.7 / 49.2	56.7 / 51.7	56.8 / 54.0
Q_{10}-Narrow	57.7 / 40.5	58.0 / 47.4	58.1 / 50.3	58.2 / 53.0	58.2 / 55.2

Table 12: Sensitivity of Rescue@4_{100} to recovery threshold τ_R across $N = 300$ tasks. $\tau_R = 0.85$ is the protocol-locked primary threshold.

Condition	$\tau = 0.75$	$\tau = 0.80$	$\tau = 0.85$	$\tau = 0.90$	$\tau = 0.95$
Q_2-Diverse	168	164	**149**	142	102
Q_5-Diverse	165	162	**151**	144	111
Q_{10}-Diverse	133	131	**126**	120	99
Q_{20}-Diverse	105	105	**105**	105	105
Q_{10}-Narrow	172	168	**153**	143	98

Table 13: Inference cost for the 300-task fresh-holdout study. Tokens/success and calls/success use Rescue@4_{100} as success criterion.

Condition	Calls/Task	Tokens/Task	Median Lat. (s)	Rescues	Tokens/Success	Calls/Success
Q_2-Diverse	2.39	3,366	9.84	149	6,777	4.81
Q_5-Diverse	2.51	3,535	9.60	151	7,022	4.99
Q_{10}-Diverse	2.77	3,897	10.85	126	9,279	6.60
Q_{20}-Diverse	3.09	4,301	11.18	105	12,288	8.84
Q_{10}-Narrow	2.41	3,417	4.89	153	6,699	4.72

Table 14: Per-family breakdown on the 300 fresh unseen tasks under Q_{20}-Diverse (50 tasks per benchmark family).

Architectural Benchmark Family	**Dev Pass**	**Hidden Pass**	**Rescue@4_{100}**	**Conservatism Gap**
Configuration (`fresh_config`)	50/50 (100.0%)	50/50 (100.0%)	50/50 (100.0%)	0/0 (0.0%)
Tools (`fresh_tools`)	15/50 (30.0%)	15/50 (30.0%)	15/50 (30.0%)	0/35 (0.0%)
Resources (`fresh_resources`)	26/50 (52.0%)	41/50 (82.0%)	26/50 (52.0%)	15/24 (62.5%)
Multi-Surface (`fresh_multi_surface`)	8/50 (16.0%)	8/50 (16.0%)	8/50 (16.0%)	0/42 (0.0%)
Middleware (`fresh_middleware`)	6/50 (12.0%)	6/50 (12.0%)	6/50 (12.0%)	0/44 (0.0%)
Listeners (`fresh_listeners`)	0/50 (0.0%)	32/50 (64.0%)	0/50 (0.0%)	32/50 (64.0%)
Total Fresh Cohort	**105/300 (35.0%)**	**152/300 (50.7%)**	**105/300 (35.0%)**	**47/195 (24.1%)**

The critical value $z = 1.96$ is the two-sided 95% normal critical value. Admission requires both split endpoints to be at least $\tau_R = 0.85$, in addition to the development and language gates. At $n = 20$, 20/20 gives 1.0000 and 19/20 gives approximately 0.85448 (both pass); 18/20 gives approximately 0.76852 (fails). Thus the hidden rule is $k_{\text{iid}} \geq 19$ and $k_{\text{ood}} \geq 19$.

The earlier Wilson description was a reporting error. A post-hoc robustness check pooled the two splits ($N = 40$) and applied the lower endpoint of the two-sided 95% Wilson interval at the same 0.85 threshold. All recorded candidate evaluations retained identical admission decisions, including 180/197 rescues under D_0L_1, with zero stratum migrations. The rules are not mathematically identical: $(19, 19)$ passes split Wald but fails pooled Wilson; that combination did not occur in the recorded candidate evaluations. Pooled Wilson was solely post-hoc rescoring of the primary protocol. The separate 100-state fresh-holdout Wilson analyses in Appendix I retain their stated scope.

J.2 Paired Contrasts

For the three protocol-locked contrasts, we use paired two-sided exact McNemar tests. Let b and c denote discordant pair counts. The exact test uses $X \sim \text{Binomial}(b + c, 0.5)$ and $p = 2\Pr[X \leq \min(b, c)]$, capped at one. Holm–Bonferroni correction is applied jointly across C_1, C_2, C_3. Paired risk-difference confidence intervals are estimated with 100,000 paired bootstrap resamples.

The final confirmatory results are:

- C_1: $\Delta = +79.17$ pp, 95% CI $[+66.67, +89.58]$, $p_{\text{raw}} = 7.28 \times 10^{-12}$, $p_{\text{Holm}} = 1.46 \times 10^{-11}$.
- C_2: $\Delta = +99.30$ pp, 95% CI $[+97.90, +100.00]$, $p_{\text{raw}} = 3.59 \times 10^{-43}$, $p_{\text{Holm}} = 1.08 \times 10^{-42}$.
- C_3: $\Delta = -6.29$ pp, 95% CI $[-11.19, -2.10]$, $p_{\text{raw}} = p_{\text{Holm}} = 0.01171875$.

K Robustness Analysis and Execution Integrity

K.1 Family-Level Leave-One-Out Sensitivity

Leave-one-family-out estimates remain strongly positive for C_1 and C_2. For C_3, the estimated risk difference remains negative across all leave-one-family-out partitions. The exploratory full-cohort factorial interaction retains its negative sign ($I = -25.89$ pp, 95% CI $[-34.01, -18.27]$).

K.2 Execution Integrity and Protocol Locking

The experimental implementation uses fail-closed execution. Model-service failures abort execution, malformed outputs consume an attempt, and recovery primitives outside the assigned language are rejected before execution. Result artifacts, task manifests, protocol configurations, success vectors, and analysis scripts are stored with cryptographic hashes to distinguish frozen result artifacts from superseded runs.