REPOT: Recoverable Program-of-Thought via Checkpoint Repair

Anonymous ACL submission

Abstract

One-shot Program-of-Thought (POT) emits a Python program that prints a primitive-action plan; a single invalid action silently invalidates the trajectory. We introduce **REPOT** (Recoverable POT): a deterministic *verified replay* that walks the plan through the environment to its first invalid transition, then one LLM call that resumes from the verified prefix. REPOT costs at most one extra LLM call on the $\sim$14% of problems where POT fails. REPOT beats POT by $+3$ to $+11$pp across four closed-model configurations on PUZZLEZOO-775 and peaks at 96.9% vs 86.3% on `gpt-5.4-mini-medium`; against the matched-budget POT-retry baseline, REPOT wins decisively on Gemini ($+3.8$ pp, 95% CI $[+2.2, +5.4]$), is within sampling noise on GPT-medium and Claude, and loses on GPT-mini — a capability-scaling pattern we begin to address with ADAPTIVE REPOT, a rule-based dispatcher that routes between suffix repair and a fresh POT retry based on verified-prefix length (preliminary; App. N). We replicate on PLANBENCH BLOCKSWORLD ($+1.1$ to $+11.4$pp) and on four open-weights models ($+3.3$ to $+20.0$pp on three of four). On DERAIL-550, our controlled recovery benchmark, every condition with access to checkpoint information clears $\geq$30% on GPT-medium and $\geq$70% on Gemini, vs $\leq$3.1% for error-only feedback — showing that *checkpoint information*, not the specific verified-prefix tail, is the load-bearing recovery signal.

1 Introduction

Large language models can sketch impressive plans, then commit one illegal action and silently fail the entire task. The dominant fix is to either run a single sample through a tool (Program-of-Thought, POT Chen et al., 2023) or sample many independent rollouts and aggregate (Self-Consistency Wang et al., 2023, Tree of Thoughts Yao et al., 2023a). Neither is recoverable: a one-shot POT plan with a mid-rollout error cannot resume from where it succeeded; tree-search methods pay branching cost during generation regardless of whether the first trajectory was already mostly correct.

We propose REPOT— *Recoverable Program-of-Thought* — a small modification of one-shot POT that adds checkpoint-based recovery without moving to tree search. REPOT works in three steps (Algorithm 1):

1. Run POT once: emit a Python program, execute it, parse the printed move list.
2. *Verified replay*: walk the proposed actions through the environment one step at a time, accumulating a maximal *verified prefix* of valid transitions until the first failure (Eq. (1)).
3. If the prefix already reaches the goal, return success. Otherwise issue *one* suffix-repair LLM call, conditioning on the verified prefix, the verified state at the failure boundary, and the verifier's error message.

Novelty and positioning. REPOT reframes one-shot LLM reasoning as a *recoverable execution*: rather than all-or-nothing, the verifier owns the trusted state and the model is only ever asked to repair the unverified suffix. Three pieces compose: (i) a deterministic verified-replay primitive that turns any POT rollout into a checkpoint-resumable computation with no LLM calls; (ii) a suffix-repair call conditioned on the *verified state* rather than a textual critique of the prior attempt — one well-typed task; (iii) a single repair budget ($R = 1$), so cost stays at POT baseline on the $\sim$86% of easy problems and only doubles on the rest. Where Reflexion-style (Shinn et al., 2023) verbal critique asks the model to introspect its mistakes textually, REPOT makes the verifier the source of truth. Where ToT and LATS (Zhou et al., 2024) branch *during* generation, REPOT branches only *after* a

deterministic check identifies a real failure point. DERAIL-550 (§7) shows the *trusted checkpoint* is the load-bearing signal: 80.7% recovery of injected errors versus 20.7% from error-only feedback.

Contributions.

- **Algorithm.** REPOT— a recoverable extension of POT that combines deterministic verified replay (Eq. 1) with a single suffix-repair LLM call.
- **Empirical.** On PUZZLEZOO-775, a verifier-backed suite, REPOT improves over POT by +3 to +11pp across three frontier models in four configurations, replicates on PLANBENCH BLOCKSWORLD (Valmeekam et al., 2023a) (+1.1 to +11.4pp), and beats a matched-budget POT-retry baseline on the two reasoning-enabled models.
- **Mechanism.** A controlled DERAIL-550 benchmark isolates which signal makes recovery work: trusted checkpoint state separates the recovery-capable conditions from error-only feedback by ∼60pp; the explicit verified-prefix tail provides a smaller, model-dependent additional benefit.

2 Background

Program-of-Thought planning. POT (Chen et al., 2023) prompts the model to write a Python program whose printed output is the plan, then runs the program in a sandbox and parses the printed token list. The original work targeted arithmetic and symbolic reasoning, where the program is the answer. In LLM planning, the printed plan is a sequence of primitive actions $(a_1, \ldots, a_n)$ which is then executed by an environment simulator, and the trajectory either reaches the goal or it does not. POT's appeal in this setting is that it pushes the brittle bookkeeping (state tracking, legal-action filtering, search) onto the model in code form, which the model is good at (Wei et al., 2022; Wang et al., 2023). The cost is also its weakness: a single illegal action mid-rollout invalidates the entire trajectory, and the trajectory's verified prefix is discarded along with the invalid suffix.

The Illusion-of-Thinking finding. Shojaee et al. (2025) ("The Illusion of Thinking") run a controlled puzzle benchmark across four classical planning environments (Tower of Hanoi, Checker Jumping, River Crossing, Blocksworld) at controllable complexity, and report that frontier reasoning models exhibit sharp accuracy collapse once complexity exceeds a model-specific threshold, even when given the algorithm. They further show that failures concentrate early in the trace: the first invalid move often appears at a small fraction of the optimal-plan length, and the remaining tokens are spent on a wrong-but-consistent continuation. Song et al. (2025) re-run a subset and argue some collapses are artifacts of impossible River Crossing instances and prompt encoding; Scholten et al. (2024) frame the same phenomenon as "metacognitive myopia". Whatever the framing, the empirical pattern is robust: most failed traces have a long valid prefix and a single mid-rollout misstep. This is precisely the regime in which checkpoint-based recovery should help. REPOT is a targeted fix for this *recoverable subset* of failures: we do not claim to solve reasoning collapse in general, only to mitigate *recoverable execution collapse* where a trusted intermediate state can be preserved and resumed from.

Prior work on one-shot brittleness. A large body of work has proposed remedies for the one-shot failure mode; we group them into three families. **(i) Sample more**: Self-Consistency (Wang et al., 2023) and best-of-k generate k trajectories and vote / pick the best by some scoring function. Cost is $O(k)$ LLM calls per problem regardless of whether the first trajectory was nearly correct. **(ii) Branch during generation**: Tree of Thoughts (Yao et al., 2023a) and LATS (Zhou et al., 2024) treat reasoning as search, expanding multiple continuations and using a value model or verifier to prune. They pay tree-search cost on every problem, including easy ones. **(iii) Iterate with critique**: Reflexion (Shinn et al., 2023), ReAct (Yao et al., 2023b), Self-Refine (Madaan et al., 2023), and Self-Repair (Olausson et al., 2024) run a second LLM call conditioned on a textual critique of the prior attempt. The critique can be wrong (*the model that produced the failure now also produces the diagnosis*), and there is no checkpoint mechanism: the second call re-plans from scratch given the critique.

Process verification and rewards. A separate line uses process reward models (PRMs) (Lightman et al., 2024) to score partial reasoning traces and reject low-scored continuations. PRMs need a learned scorer and target mathematical reasoning where ground-truth verification is hard. REPOT sits in the complementary regime where the verifier is the environment itself — exact, free, and imme-

diately available — which lets us skip the learned scorer entirely.

3 Related Work

Section 2 surveyed the main families that REPOT relates to. Here we position REPOT against three further adjacencies. **Concurrent transactional/checkpoint work** (Mohammadi et al., 2026; Chang and Geng, 2025; Li et al., 2025) targets multi-agent coordination rather than one-shot POT. **Decoupled reasoning–observation** (ReWOO, Xu et al., 2023; ThinkSwitcher, Liang et al., 2025) routes between thinking modes per input; REPOT routes per-trajectory, conditioned on a verified failure boundary. **Planning benchmarks** PLANBENCH (Valmeekam et al., 2023a,b) provides PDDL-grounded LLM planning evaluation; we use its Blocksworld split as external replication and discuss the agentic-gap framing (Khan et al., 2025) in Discussion.

4 Method

4.1 Problem setting

We consider planning tasks of the form (s_0, g, E), where s_0 is the initial state, g is the goal specification, and E is a deterministic environment with a step function $E : (s, a) \mapsto (s', \text{valid})$ that returns the next state and a validity flag. A plan is a sequence of primitive actions $\pi = [a_1, a_2, \ldots, a_n]$. Success is $\textsc{IsGoal}(s_n, g)$ where s_n is the result of replaying π from s_0. We assume the environment exposes a verifier (the same STEP) and a goal predicate; we do not assume the model has direct access to either.

4.2 Verified replay

The core primitive is a deterministic *verified-replay* function that takes a candidate action sequence and walks it through the environment. Given start state s_0 and a plan $\pi = (a_1, \ldots, a_n)$, define the trajectory $s_i = \text{step}(s_{i-1}, a_i)$ if the transition is valid, else $\perp$. Let k be the smallest index for which the transition is invalid (or $n + 1$ if every action is valid). Then

$$\textsc{Replay}(E, s_0, \pi) \;=\; \big(P,\; s_{k-1},\; \epsilon_k\big), \qquad (1)$$

where $P = (a_1, \ldots, a_{k-1})$ is the maximal verified prefix, s_{k-1} is the verified state at the failure boundary, and ϵ_k is the verifier's error message at step k (or the empty string if $k = n + 1$). The function is total, deterministic, and makes no LLM calls; its cost is $O(n)$ environment steps.

Algorithm 1 REPOT— Recoverable Program-of-Thought

Input: problem (s_0, g), env E, model $\mathcal{M}$, repair budget R

$\pi \leftarrow \mathcal{M}\big(\textsc{PoTPrompt}(s_0, g)\big)$ ▷ 1. one-shot POT
$(P, s, \epsilon) \leftarrow \textsc{Replay}(E, s_0, \pi)$ ▷ 2. verified replay (Eq. 1)
if $\textsc{IsGoal}(s, g)$ **then return** P
end if
for $r = 1, \ldots, R$ **do** ▷ 3. suffix repair from checkpoint
  $\pi' \leftarrow \mathcal{M}\big(\textsc{RepairPrompt}(s_0, g, s, P, \epsilon)\big)$
  $(P', s, \epsilon) \leftarrow \textsc{Replay}(E, s, \pi')$
  $P \leftarrow P \mathbin{+\!\!+} P'$
  if $\textsc{IsGoal}(s, g)$ **then return** P
  end if
end for
return P ▷ partial; budget exhausted

4.3 REPOT algorithm

REPOT composes POT with verified replay and a single suffix-repair call. The full algorithm is shown in Algorithm 1. Two hyperparameters govern its behaviour: the repair budget R (default $R{=}1$) and the verified-prefix tail length T shown to the model during repair (default $T = 4$). The model is otherwise sampled at $\tau{=}0$.

A simple recovery model. For a problem instance, let p be the probability that POT succeeds on the first sample, q the probability that POT fails but leaves a recoverable valid prefix, r the conditional probability that suffix repair succeeds given a recoverable prefix, b the conditional probability that a fresh POT resample succeeds given the first sample failed, and $b' \leq b$ the fresh-retry success rate restricted to the unrecoverable subset. REPOT beats POT-retry iff

$$q\,(r - b) \;>\; (1 - p - q)\,(b - b'), \qquad (2)$$

i.e. when verified-prefix repair beats the fresh-sample marginal on the recoverable subset. Larger q (longer valid prefixes, scaling with capability) makes the condition more favourable; §6.4 and Fig. 3 show this empirically. The adaptive variant below dispatches per-problem to maximize Eq. 2.

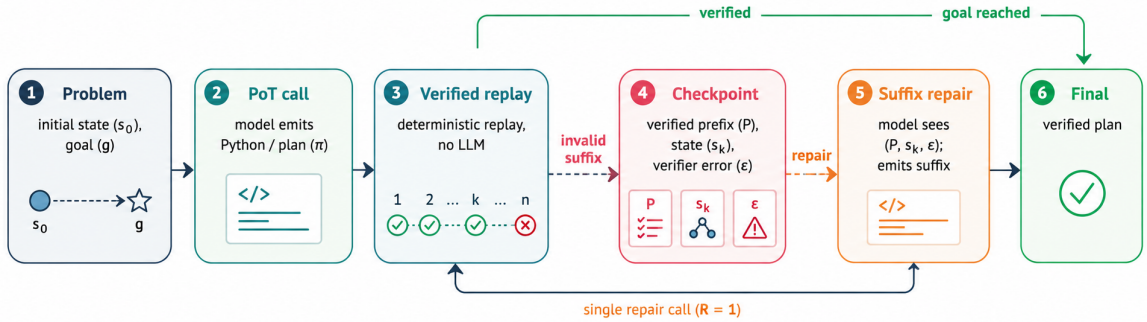

Figure 1: The REPOT pipeline. **(1) Problem** provides initial state s_0 and goal g. **(2) POT call** (LLM call #1): the model emits a Python program whose stdout encodes the action plan π. **(3) Verified replay**: walk π through the environment one step at a time — deterministic, no LLM calls — producing the maximal valid prefix and the failure boundary. If every action is valid, branch to **(6) Final** via the upper *verified, goal reached* arrow. Otherwise the boundary is packaged as **(4) Checkpoint**: verified prefix P, verified state s_k, and verifier error ε. The **(5) Suffix repair** call (LLM call #2) sees the checkpoint and emits a suffix that resumes from s_k. The lower arrow is the single repair budget ($R=1$): the suffix is itself sent through verified replay; on success we land at Final, on failure we accept the partial verified plan.

4.4 Adaptive recovery policy

Algorithm 1 always commits to the verified prefix. When the prefix is empty or very short, the verified state s collapses to s_0 and the repair call effectively restarts from the initial state but with a (potentially misleading) error anchor. Empirically, on weaker models POT-retry's fresh sample outperforms anchoring on a short prefix (§6.1).

As a preliminary extension, we introduce **ADAPTIVE REPOT**, a rule-based dispatcher with the same $R=1$ budget. After verified replay, we read the prefix fraction $\phi = (k-1)/n$ and route to a *fresh* POT *retry* when $n = 0$ or $\phi < 0.15$, otherwise to suffix repair (Alg. 1). Thresholds were fixed a priori (not tuned on test); a threshold sweep and alternative dispatcher rules are left to future work. The dispatcher realizes the optimal-branch prediction implied by Eq. 2: route to fresh sampling when the recoverable subset is empty, otherwise exploit the verified prefix. Open-source results are in App. N.

4.5 Repair prompt and ablations

The repair call uses a verified-prefix-conditioned prompt: a stable block (problem statement, goal) above a verifier-checkpoint marker, and a dynamic block (last T verified moves, current verified state, legal actions, verifier error) below. Splitting along this boundary makes the stable block prefix-cacheable across repair calls; the full template is in App. K (Fig. 12).

Three named ablations are used in §7: **REPOT** $_{\text{full}}$ (Algorithm 1), **REPOT** $_{\text{no-prefix}}$ (hides the prefix tail; model sees only the current verified state + error), and **REPOT** $_{\text{restart}}$ (repair call restarts from s_0 instead of the verified s, isolating "checkpoint" from "extra call"). Definitions in App. L.

5 Experimental Setup

5.1 Benchmarks

PUZZLEZOO-775. A stratified problem set across four classical planning environments: Tower of Hanoi (8 complexities, 200 problems), Checker Jumping (9 complexities, 225 problems), River Crossing (4 complexities, 100 problems), and Blocksworld (10 complexities, 250 problems). Each environment exposes STEP, ISGOAL, NORMALIZE, and LEGALACTIONS interfaces.

PLANBENCH Blocksworld (378 problems). We use both subsets of PLANBENCH's Blocksworld split (Valmeekam et al., 2023a): `generated_basic` (189 4-block instances) and `generated` (189 instances spanning 3–12 blocks). We adapt the PDDL semantics into our STEP interface (predicate state, four-op action vocabulary {pick-up, put-down, stack, unstack}, partial-goal subset check); see Appendix H for the adapter.

DERAIL-550 (550 errors × 11 conditions). A controlled recovery-from-injected-error benchmark we build alongside PUZZLEZOO-775. For each problem, we run an oracle plan to a checkpoint $\sim 1/3$ of the way through, inject one randomly chosen wrong action, and ask each recovery method to take over. We compare 11 conditions (§7) including REPOT's three prefix ablations.

5.2 Models

We evaluate REPOT on both closed-source and open-weights models. The closed set comprises three frontier models in four configurations: `gpt-5.4-mini-medium` (reasoning medium), `gpt-5.4-mini` (no reasoning), `gemini-3.5-flash` (thinking=MEDIUM), and `claude-sonnet-4.6` (no thinking). The open-weights set comprises four models served on a single NVIDIA H100 80GB GPU via vLLM with extended thinking disabled: `Qwen3.6-35B-A3B` (Qwen Team, 2026), `gemma-4-26B-A4B-it` (Google DeepMind, 2025), `gpt-oss-20b` (OpenAI, 2025), and `Nemotron-3-Nano-30B-A3B` (NVIDIA, 2025). Sampling is deterministic ($\tau = 0$); full hyperparameters are in Table 4 (App. B).

5.3 Methods compared

COT, POT, Self-Consistency (SC, $k = 8$), POT-retry, and our REPOT ($R=1, T=4$). COT and SC emit prose plans; POT, POT-retry, and REPOT all emit Python code. POT-retry is a matched-budget control: run POT once, and on verifier failure run POT once more from scratch with no prefix and no checkpoint — the same two-LLM-call worst-case budget as REPOT, but with no checkpoint mechanism. The POT vs POT-retry vs REPOT triple lets us separate re-rolling from genuine checkpoint-based recovery.

6 Results

6.1 Headline cross-model accuracy

Table 1 reports success rate on PUZZLEZOO-775 for the four closed models. REPOT beats POT on every model with $\Delta \in [+2.7, +10.6]$pp; the largest improvement is on `gpt-5.4-mini-medium` (96.9% vs 86.3%). Against the matched-budget POT-retry baseline, REPOT wins decisively on Gemini (+3.8, 95% CI $[+2.2, +5.4]$), is statistically a tie on GPT-medium and Claude (CIs cross zero), and loses on GPT-mini (-6.6, $[-9.4, -3.7]$). The result is consistent with our central claim that REPOT's mechanism contribution scales with the validity of the first POT plan, which itself scales with model capability (§8). Per-method cost is in App. G (Fig. 9).

6.2 Per-environment breakdown

The REPOT $-$POT delta concentrates in Blocksworld (+9.2 to +17.2pp on every reasoning model) and Checker Jumping (up to +15.6 on `gpt-5.4-mini-medium`). Hanoi and River Crossing are saturated by POT on most models ($\geq 99\%$); the exception is `gemini`'s River Crossing collapsing to 76% POT, which REPOT recovers to 100% (+24.0pp). Per-environment numbers are in Appendix A (Table 5).

6.3 External replication: PLANBENCH BLOCKSWORLD

We repeat the POT vs. REPOT comparison on PLANBENCH BLOCKSWORLD (378 instances, 3–12 blocks). To keep cost low and avoid POT saturating, we use the no-thinking variants of each model. Table 2 reports the headline; the per-complexity breakdown is in Appendix A. REPOT improves over POT on all three models. The biggest gains land in the mid-complexity band (4–6 blocks), where POT has both failure headroom and enough structure to recover toward.

Limitation: matched-budget control. The PLANBENCH comparison reports POT vs REPOT only; we do not run POT-retry on PLANBENCH in this version. The external replication therefore validates only the raw lift, not the matched-budget claim of Table 1.

Multi-seed variance. REPOT $-$ POT is positive on every seed for all three reasoning-thinking-on configurations ($\Delta \in [+1, +10]$pp); per-seed numbers and lower run-to-run variance under REPOT are tabulated in Table 6 (App. D).

6.4 Open-source replication and capability scaling

We replicate the headline comparison on a 120-problem stratified subset with four open-weights models served via vLLM with extended thinking disabled. Per-model rates appear in the bottom block of Table 1 and visually in Fig. 2; REPOT improves over POT on three of four open-source models (+3.3 to +20.0 pp). Nemotron-3 Nano 30B FP8 underperforms POT by 15 pp; with a COT baseline of 6.7% it sits near the instruction-following floor for this task family, the predicted failure mode of Eq. 2 when per-recoverable repair success r collapses. As a preliminary extension, ADAPTIVE REPOT (App. N) further closes the gap to POT-retry on the weaker rows.

Beyond per-model numbers, the open-source spread lets us test the prediction of Eq. 2 quantitatively. Across (model, environment) cells, the

Model	CoT	$SC_{k=8}$	PoT	PoT-retry	**RePoT**	R−PR
Closed frontier models						
GPT-5.4-mini (med)*	11.6	37.5	86.3	96.6	**96.9**	+0.3
Gemini 3.5 Flash*	83.0	96.6	81.3	84.1	**87.7**	+3.6
Claude Sonnet 4.6	44.1	75.9	83.1	**87.5**	86.1	−1.4
GPT-5.4-mini	17.9	24.4	58.7	**68.0**	61.4	−6.6
Open-source models						
Gemma 4 26B-A4B	18.3	24.2	49.2	54.2	**69.2**	+15.0
GPT-OSS 20B	31.7	58.3	48.3	59.2	**62.5**	+3.3
Qwen 3.6 35B-A3B	36.7	59.2	55.0	**61.7**	58.3	−3.4
Nemotron-3 Nano 30B	6.7	8.3	25.8	**34.2**	10.8	−23.4

Table 1: Cross-model success rate (%). R−PR is RePoT−PoT-retry in percentage points. PoT-RETRY is a matched-budget control: run PoT, on failure run PoT a second time from scratch (no checkpoint, no prefix). Paired bootstrap 95% CIs on R−PR ($B = 10000$). *Closed:* GPT-med $[-1.4, +1.9]$, Gemini $[+2.2, +5.4]$, Claude $[-3.1, +0.3]$, GPT-mini $[-9.4, -3.7]$ — Gemini and GPT-mini are significant; GPT-med and Claude are within sampling noise. *Open-source:* Gemma $[+4.2, +25.0]$, GPT-OSS $[-6.7, +13.3]$, Qwen $[-10.8, +4.2]$, Nemotron $[-31.7, -15.0]$ — Gemma significant positive, Nemotron significant negative, GPT-OSS and Qwen within sampling noise. REPOT's contribution scales with the validity of the first PoT plan, which itself scales with model capability (§8); ADAPTIVE REPOT is in App. N. *reasoning/thinking enabled at `medium`; unmarked closed rows run without reasoning.

Model	n	PoT	RePoT	Δ
GPT-5.4-mini	756	55.0	**64.3**	+9.3
Claude Sonnet 4.6	756	79.9	**91.3**	+11.4
Gemini 3.5 Flash	756	98.9	**100.0**	+1.1

Table 2: External replication on PLANBENCH BLOCKSWORLD (378 instances, 3–12 blocks). Success rate (%); $n = 756$ records per row (378 problems $\times$ 2 methods). **All three models run without thinking** (no reasoning effort on GPT-5.4-mini; thinking off on Claude Sonnet 4.6; `thinking_level=NONE` on Gemini 3.5 Flash) to keep cost low and avoid PoT saturation. **RePoT beats PoT on every model.** Paired bootstrap 95% CIs ($B = 10000$) on Δ: GPT-mini $[+3.5, +14.9]$, Claude $[+6.9, +15.9]$, Gemini $[+0.3, +2.1]$ — all three significant. PoT-RETRY is not run on PLANBENCH in this version (limitation; the matched-budget claim of Table 1 is not externally replicated here).

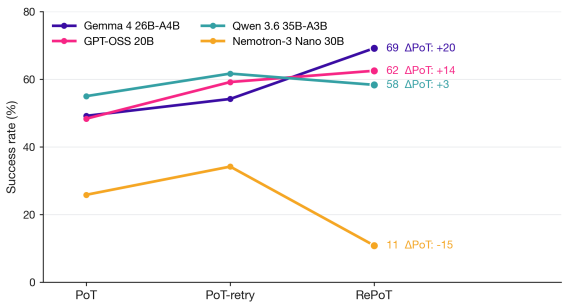

Figure 2: Open-source replication. REPOT lift tracks model capability: Gemma 4 (top) gains +20 pp over PoT; Nemotron-3 Nano 30B FP8 (right) is the predicted capability-floor failure (Eq. 2).

mean verified-prefix fraction on failed initial PoT plans (a model-level proxy for q in Eq. 2) correlates positively with the REPOT −PoT-retry success-rate delta: cells where the model leaves a long valid prefix before failing are exactly the cells where REPOT's verified-prefix repair beats fresh resampling (Fig. 3, slope +35). This is the central qualitative claim of the paper made quantitative.

7 Mechanism Analysis

7.1 Checkpoint information is the load-bearing signal

DERAIL-550 compares 11 recovery methods on 550 injected errors per model on two reasoning-thinking-on configurations (Gemini, GPT (med)). The decisive mechanism evidence is the gap between conditions that see *checkpoint information* (verified state s, legal actions, and the verifier error ε) and conditions that see only an error message: every checkpointed method clears 30% on GPT (med) and 70% on Gemini, while `error_only` stays at 3.1% / 20.7% and `no_feedback` at 1.6% / 3.8%. The ∼30–80pp gap is the headline finding: **checkpoint information — not textual error feedback or the specific verified-prefix tail — is the decisive recovery signal** (Fig. 4, Table 3). Within the checkpointed conditions, the within-prefix ablation is mixed: REPOT $_{\text{full}}$ beats REPOT $_{\text{no-prefix}}$ by +3.8 / +5.8pp on Gemini / GPT (med), but REPOT $_{\text{restart}}$ (which discards the verified state and restarts from s_0 with the same checkpoint information) beats REPOT $_{\text{full}}$ on both models, with a large gap on GPT (med) (94.5 vs 59.6). We read

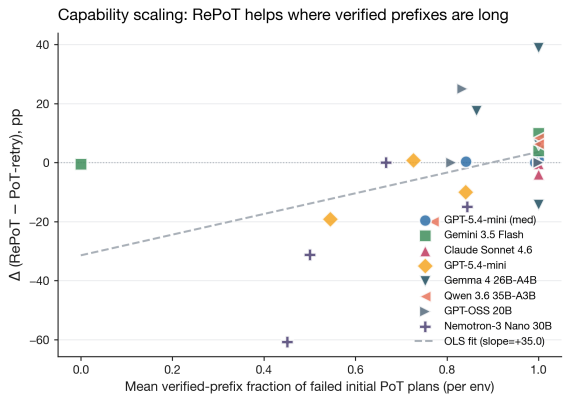

Figure 3: Capability scaling. Each point is one (model, environment) cell across both closed and open-source runs. x: mean verified-prefix fraction on failed initial PoT plans (a model-level proxy for q in Eq. 2); y: RePoT −PoT-retry success delta in pp. The positive slope confirms the mechanism: RePoT's lift scales with the validity of the first PoT plan, which itself scales with model capability.

Recovery condition	Gemini	GPT (med)
No-checkpoint baselines		
no_feedback	3.8	1.6
error_only	20.7	3.1
Checkpointed baselines		
state_feedback	**83.1**	40.4
state_plus_legal_actions	78.5	37.6
stateguard_rollback	13.1	4.5
RePoT prefix-conditioning ablation (ours)		
repot_full	80.7	59.6
repot_no_prefix	76.9	53.8
repot_restart	**82.4**	**94.5**

Table 3: DERAIL-550: success rate (%) on 550 injected errors per condition, on two reasoning-thinking-on configurations (Gemini = Gemini 3.5 Flash with `thinking_level=MEDIUM`; GPT (med) = GPT-5.4-mini with reasoning at `medium`). **Headline:** a ∼30–80pp gap separates every *checkpointed* condition (`state_feedback`, `repot_*`) from non-checkpointed baselines (`error_only`, `no_feedback`: $\leq 3\%$ on GPT (med), $\leq 21\%$ on Gemini). **Trusted checkpoint state is the load-bearing recovery signal**, not textual error feedback alone. RePoT operationalises that signal inside PoT via verified replay. The within-RePoT trio shows `repot_restart`>`repot_full` on both models (more pronounced on GPT (med)) — the prefix-tail beyond the checkpoint is a model-dependent secondary effect, discussed in §7.1 and App. J.

this honestly: anchoring the repair on the verified-prefix tail beyond the checkpoint can hurt, especially on weaker configurations; the checkpoint information itself is what matters, not the specific resumption point. ADAPTIVE REPOT's dispatcher operationalizes this: short prefixes route to fresh retry, only long prefixes anchor on the verified state.

Cost. REPOT averages 1.11–1.39× PoT LLM calls across the four configurations. Per-method, per-model breakdown in Table 8 (App. G).

Failure modes. The dominant REPOT failure is `repair_budget_exhausted` on hard Blocksworld; most hand-analyzed losses involve an empty initial PoT plan with no prefix to anchor on (App. I).

8 Discussion

When REPOT helps and when it does not. REPOT's lift concentrates where PoT produces a long valid prefix before failing: a mostly-correct plan whose suffix needs repair. Where PoT already succeeds (saturated easy regimes, `gemini` on Hanoi at 99.5%) REPOT is no-op. Where PoT fails at the first action, REPOT's replay degenerates to a restart-from-s_0 and lift is near zero. This shape predicts the matched-budget PoT-retry result in Table 1: on the two reasoning-enabled models (`gpt-medium`, `gemini`) REPOT beats PoT-retry by +0.3 and +3.6pp; on the two non-reasoning rows (`claude` no-thinking, `gpt-mini` no-reason) PoT-retry beats REPOT by 1.4 and 6.6pp. We read this as a scoped claim: REPOT is the right move when the model is capable enough to produce useful valid prefixes; when it is not, a fresh independent sample (PoT-retry) escapes wrong commitments more effectively.

Future work. Empty-plan retry to close hand-analyzed losses; adaptive repair budget conditioned on the verified-prefix fraction; open-source replication; broader benchmarks (PDDLGym, ALFWorld).

9 Conclusion

REPOT is a small, structural addition to PoT: deterministic verified replay plus one bounded suffix-repair call. Against the matched-budget PoT-retry baseline on PUZZLEZOO-775, REPOT wins decisively on Gemini (CI [+2.2, +5.4]), is within sampling noise on GPT-medium and Claude, and loses on GPT-mini — a capability-scaling pattern that DERAIL-550 isolates: *checkpoint information*, not the specific verified-prefix tail, is the load-bearing recovery signal. The cost is one extra LLM call on the ∼14% of problems where PoT fails the first time; the rest run at PoT cost. REPOT is

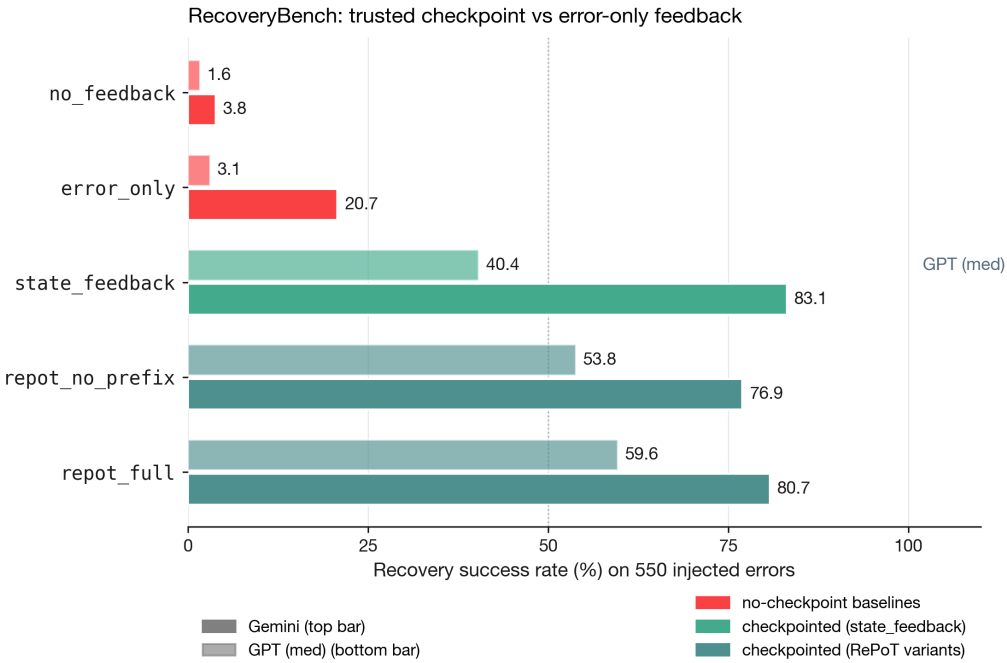

Figure 4: DERAIL-550, headline conditions only. The ∼60pp gap between checkpointed and no-checkpoint conditions is the load-bearing finding. The full 11-condition table is in Table 3.

the cheapest viable middle between one-shot PoT and tree-search-based methods for verifier-backed planning, and a candidate primitive for any setting where the environment exposes a deterministic step function and a goal predicate. Generalisation to verifier-backed agentic settings (coding, SQL, browser automation) is left to future work.

Limitations

Verifier-required scope. REPOT assumes a deterministic verifier (puzzles, planning, tool use); free-form reasoning would need a learned scorer (Lightman et al., 2024). The evaluation in this paper is restricted to verifier-backed puzzle and planning environments (PUZZLEZOO-775, PLANBENCH BLOCKSWORLD, DERAIL-550). Whether the same recoverable-execution abstraction generalizes to verifier-backed agentic settings (coding agents with test suites, SQL agents with schema checks, browser automation with page-state validators) is an open empirical question we do not address.

Single-call repair budget. We use $R = 1$ as a cost-conservative choice; multi-call repair (Alg. 1 line 4) likely closes the long-horizon BLOCKSWORLD tail but is not evaluated here.

Within-prefix conditioning is model-dependent. On DERAIL's controlled mid-rollout error setting, REPOT $_{\text{restart}}$ beats REPOT $_{\text{full}}$ on both configurations (more pronounced on GPT (med), 94.5 vs 59.6). The checkpoint-state contribution remains the load-bearing recovery signal, but anchoring the repair on the verified-prefix tail can hurt on weaker configurations (App. J).

Negative-Δ cells in PLANBENCH. GPT (mini) at $c = 8, 11$ shows -8.3 and -9.1pp REPOT vs PoT, where single-call repair sometimes commits to a bad prefix at low PoT-base accuracy.

Statistical scope. PLANBENCH reports PoT vs REPOT only; we do not run PoT-retry on PLANBENCH in this version, so the matched-budget claim of Table 1 is not externally replicated. Open-source results are on a 120-problem stratified subset, smaller than the closed-model headline.

Adaptive RePoT is preliminary. ADAPTIVE REPOT uses a hand-picked threshold ($\phi < 0.15$); a sensitivity sweep over ϕ and alternative dispatcher rules (e.g., learned gating) are left to future work.

References

Edward Y. Chang and Longling Geng. 2025. SagaLLM: Context management, validation, and transaction

guarantees for multi-agent llm planning. *Preprint*, arXiv:2503.11951.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2023. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Transactions on Machine Learning Research (TMLR)*.

Google DeepMind. 2025. Gemma 4: Open multimodal models. Model card; Apache 2.0 license.

Sheraz Khan, Subha Madhavan, and Kannan Natarajan. 2025. A comment on "the illusion of thinking": Reframing the reasoning cliff as an agentic gap. *Preprint*, arXiv:2506.18957.

Peiran Li, Xinkai Zou, Zhuohang Wu, Ruifeng Li, Shuo Xing, Hanwen Zheng, Zhikai Hu, Yuping Wang, Haoxi Li, Qin Yuan, Yingmo Zhang, and Zhengzhong Tu. 2025. SafeFlow: A principled protocol for trustworthy and transactional autonomous agent systems. *Preprint*, arXiv:2506.07564.

Guosheng Liang, Longguang Zhong, Ziyi Yang, and Xiaojun Quan. 2025. Thinkswitcher: When to think hard, when to think fast. In *Findings of the Association for Computational Linguistics: EMNLP 2025*.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2024. Let's verify step by step. In *International Conference on Learning Representations (ICLR)*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-Refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Bardia Mohammadi, Nearchos Potamitis, Lars Klein, Akhil Arora, and Laurent Bindschaedler. 2026. Atomix: Timely, transactional tool use for reliable agentic workflows. *Preprint*, arXiv:2602.14849.

NVIDIA. 2025. Nemotron 3 Nano: Open, efficient mixture-of-experts hybrid Mamba–Transformer model for agentic reasoning. *Preprint*, arXiv:2512.20848. Technical report.

Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2024. Is self-repair a silver bullet for code generation? In *International Conference on Learning Representations (ICLR)*.

OpenAI. 2025. gpt-oss-120b & gpt-oss-20b model card. *Preprint*, arXiv:2508.10925.

Qwen Team. 2026. Qwen3.6-35B-A3B: Agentic coding power, now open to all.

Florian Scholten, Tobias R. Rebholz, and Mandy Hütter. 2024. Metacognitive myopia in large language models. *Preprint*, arXiv:2408.05568.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Parshin Shojaee, Iman Mirzadeh, Keivan Alizadeh, Maxwell Horton, Samy Bengio, and Mehrdad Farajtabar. 2025. The illusion of thinking: Understanding the strengths and limitations of reasoning models via the lens of problem complexity. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Zhao Song, Song Yue, and Jiahao Zhang. 2025. Thinking isn't an illusion: Overcoming the limitations of reasoning models via tool augmentations. *Preprint*, arXiv:2507.17699.

Karthik Valmeekam, Matthew Marquez, Alberto Olmo, Sarath Sreedharan, and Subbarao Kambhampati. 2023a. PlanBench: An extensible benchmark for evaluating large language models on planning and reasoning about change. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Karthik Valmeekam, Matthew Marquez, Sarath Sreedharan, and Subbarao Kambhampati. 2023b. On the planning abilities of large language models – a critical investigation. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. In *International Conference on Learning Representations (ICLR)*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Binfeng Xu, Zhiyuan Peng, Bowen Lei, Subhabrata Mukherjee, Yuchen Liu, and Dongkuan Xu. 2023. ReWOO: Decoupling reasoning from observations for efficient augmented language models. *Preprint*, arXiv:2305.18323.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023a. Tree of thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023b. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2024. Language agent tree search unifies reasoning, acting, and planning in language models. In *International Conference on Machine Learning (ICML)*.

A Per-environment success vs complexity (curves)

The main-text per-environment summary is the table-form Δ in Table 5. Figure 5 shows the same data as a colour-coded heatmap, and Figure 6 shows the underlying success-rate curves vs complexity for `gpt-5.4-mini-medium`, the strongest reasoning configuration.

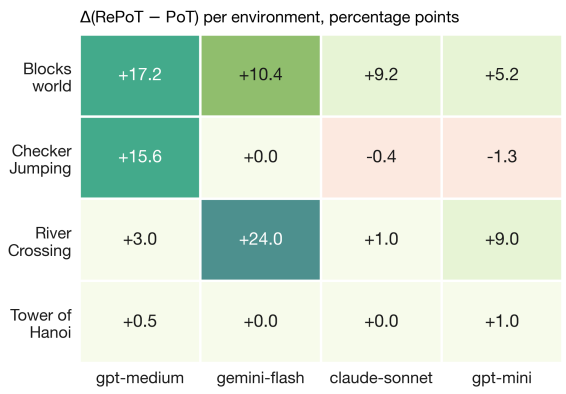

Figure 5: Δ(RePoT−PoT) per environment, in percentage points, for all four models. Blocksworld is REPOT's home environment (+5 to +17pp on every model); Hanoi/Checker on strong models are saturated by POT at small N.

B Hyperparameters

C Data generation and the controller architecture

PUZZLEZOO-775 ($n = 775$). Figure 7 shows one example instance for each of the four classical planning environments we use, illustrated at small complexity. We generate problems with controllable complexity across Tower of Hanoi, Checker Jumping, River Crossing, and Blocksworld, in the spirit of Shojaee et al. (2025). We do not redistribute Shojaee et al.'s instances; we generate fresh problems following the same families and the same controllable-complexity protocol.

For each environment we generate problems across a complexity range (e.g. Tower of Hanoi $N \in [2, 14]$, Blocksworld $N \in [3, 16]$ blocks). For every problem we additionally generate an oracle plan from a domain solver (BFS for Hanoi/Checker, brute-force enumeration for small River Crossing, FF-style stack reasoning for Blocksworld) and verify both that the oracle solves the instance and that the instance is not trivially solved by greedy heuristics. Instances that fail either check are discarded. The released suite contains 775 verified problems stratified across environment $\times$ complexity, in JSONL format.

Instance schema. Each row carries: `problem_id`, `environment`, `complexity`, an explicit `initial_state` and `goal_state` (or goal predicate), an `oracle_plan` (action list, lower-bound length), the `oracle_plan_length`, and a `natural_language_prompt` that the model sees. Each environment supplies its own verifier and goal predicate.

The controller. All methods (COT, POT, SC, REPOT) run through a common inference-time controller (the `Runner` object). The controller mediates every interaction between the LLM and the environment: it constructs the prompt, calls the LLM, parses the output, then either submits the entire plan to the environment (POT, SC, COT) or steps actions one-at-a-time through the verifier (REPOT). This mirrors the pattern advocated by Shojaee et al. (2025) and Scholten et al. (2024) of externalising state from the model's hidden reasoning into an authoritative simulator. The controller logs:

- the verbatim model prompt and completion (`prompt`, `output_text`) for every LLM call;
- per-problem method metadata: number of repair calls (`repot_repair_calls`), whether the initial POT inside REPOT succeeded (`repot_initial_pot_success`), the verified-prefix length and total plan length, the action where the first invalid transition was detected, and the verifier's error message at that boundary;
- end-to-end latency, prompt and completion token counts, and runtime exceptions if any (the `runner_exception` field, used to filter poisoned runs as described in Appendix J).

A failed LLM call (network, JSON parse, sandbox timeout) is recorded as a `runner_exception` and the problem is treated as unsolved for that method; we do not re-roll. The choice keeps the per-method denominator stable across methods on the same problem.

Verifier semantics. The verifier is the environment's transition function; for each proposed action a at state s it returns $(s', \text{ok}, \epsilon)$: s' is the next state, ok is a boolean validity flag, and ϵ is a one-line

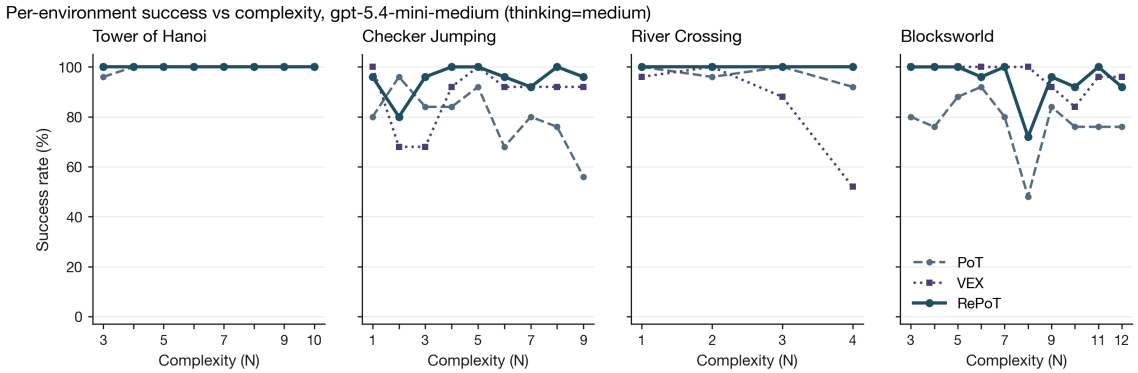

Figure 6: Per-environment success rate vs problem complexity for `gpt-5.4-mini-medium` (thinking=medium). Tower of Hanoi and Checker Jumping are nearly saturated by PoT at small N; RePoT's lift concentrates on Blocksworld and the mid-complexity dips of Checker and River.

Hyperparameter	Value	Notes
RePoT		
Repair budget R	1	one suffix-repair call
Verified-prefix tail T	4	last 4 valid moves shown to model
Sampling (all methods)		
Temperature	0.0	deterministic
`max_tokens` (output)	16,384	per LLM call
Self-Consistency baseline		
k (samples)	8	majority vote on full plans
Models		
`gpt-5.4-mini-medium`	reasoning=medium	OpenAI Responses API
`gpt-5.4-mini`	reasoning=none	OpenAI Responses API
`gemini-3.5-flash`	thinking_level=MEDIUM	Vertex AI
`gemini-3.5-flash` (PlanBench)	thinking_level=NONE	no-thinking variant
`claude-sonnet-4.6`	no extended thinking	Anthropic API

Table 4: All hyperparameters used in this paper. Each row appears verbatim in our run configs; no tuning per-method beyond what is listed. Defaults follow the public PuzzleZoo-775 config.

natural-language error message used by RePoT's repair prompt and by DeRail's `error_only` condition. Goal achievement is checked by a problem-specific `is_goal` predicate that accepts partial-goal specifications (e.g. Blocksworld goal stacks need not name every block).

Why an interactive controller is the right framing. Shojaee et al. (2025)'s key empirical pattern is that failures concentrate early in the trace, and the model spends the rest of the budget elaborating a wrong-but-consistent continuation. A one-shot PoT setup makes this invisible: the only signal is a final pass/fail. A one-action-at-a-time controller exposes the failure boundary directly (the `first_failure_move_id`, in their notation), which is exactly what RePoT's verified-replay primitive depends on. The controller is therefore not just a runtime convenience: it is the substrate that makes verified-prefix recovery well-defined.

Environment	GPT (med)	Gemini	Claude	GPT (mini)
Blocksworld	**+17.2**	**+10.4**	**+9.2**	**+5.2**
Checker Jumping	**+15.6**	+0.0	-0.4	-1.3
River Crossing	+3.0	**+24.0**	+1.0	**+9.0**
Tower of Hanoi	+0.5	+0.0	+0.0	+1.0

Table 5: Δ(RePoT$-$PoT) per environment, in percentage points. **Bold** cells have $|\Delta| \geq 5$pp. Blocksworld is RePoT's home environment; Hanoi/Checker on the strongest models are saturated by PoT. Per-model thinking config matches Table 1.

D Per-method, per-env, per-complexity full table

The full per-environment, per-complexity success rates for all 4 models $\times$ 5 methods on PuzzleZoo-775 are released as part of the supplementary materials; the headline summary is in Table 1 and the per-environment Δ in Table 5.

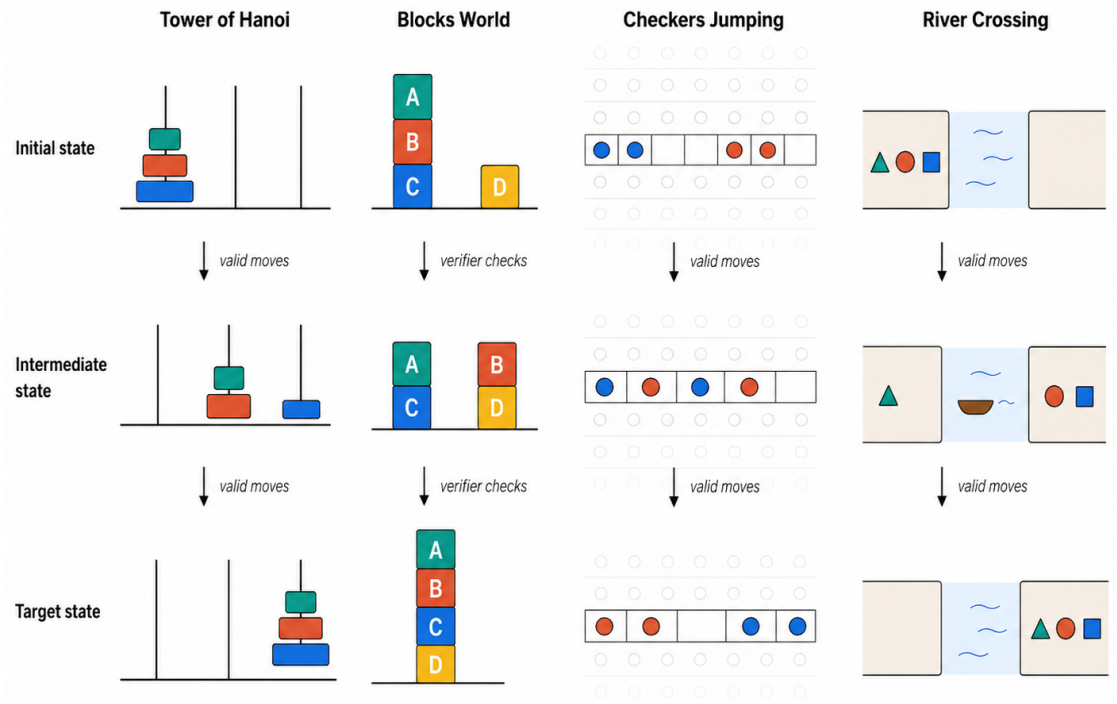

Figure 7: One example per environment at small complexity. Each column is one environment (Tower of Hanoi, Checker Jumping, River Crossing, Blocksworld); rows show *initial state*, an *intermediate state* after a few valid actions, and the *target state*. Vertical arrows mark the per-step *valid moves* that the verifier accepts. Each environment exposes a deterministic step function $E.\text{step}(s, a) \rightarrow (s', \text{ok}, \varepsilon)$ used by both the REPLAY primitive (Eq. 1) and the DERAIL harness; complexity is controlled by the integer parameter named in column headers (# disks for Hanoi, # blocks for Blocksworld, # checkers for Checker Jumping, # pairs for River Crossing).

Model	Seed	PoT	RePoT	Δ
`gpt-5.4-mini-medium`	1	86.0	96.0	+10.0
	2	85.0	94.0	+9.0
	3	88.0	95.0	+7.0
	mean	86.3	95.0	+8.7
`claude-sonnet-4.6`	1	79.0	88.0	+9.0
	2	81.0	88.0	+7.0
	3	85.0	88.0	+3.0
	mean	81.7	88.0	+6.3
`gemini-3.5-flash`	1	80.0	85.0	+5.0
	2	82.0	83.0	+1.0
	3	82.0	84.0	+2.0
	mean	81.3	84.0	+2.7

Table 6: Multi-seed paired comparison ($n = 100$ stratified problems per seed, three seeds per model). REPOT − POT is positive on every seed of every model.

Multi-seed raw numbers. Per-seed paired comparisons of REPOT vs POT on the three reasoning-thinking-on configurations are reported in Table 6. REPOT − POT is positive on every seed for all three configurations.

E Paired mechanism analysis

To isolate the verified-prefix mechanism from API sampling noise on trivial problems, we restrict to the matched-difficulty subset where both POT-retry's attempt-1 and REPOT's initial POT call failed. On this subset we compare two strategies that share the same matched budget: REPOT's verified-prefix suffix repair, and POT-retry's fresh independent sample. Table 7 reports the result and Fig. 8 shows the stacked-bar decomposition. The pattern is heterogeneous: REPOT recovers +14.3pp more than fresh resampling on Gemini, is within noise on Claude, and loses on the two GPT configurations. The pattern tracks capability scaling (§6.4): REPOT's mechanism contribution concentrates on configurations where the initial POT plan leaves a useful valid prefix, which is precisely the heterogeneity the adaptive policy (§4.4) is designed to absorb.

F Multi-candidate "best-in-thought" analysis

We re-extract the highest-success candidate from each POT reasoning trace (Apple-paper proto-

Model	N	RePoT%	Retry%	Δ
GPT-5.4-mini (med)	23	60.9	73.9	−13.0
Gemini 3.5 Flash	112	17.9	3.6	+14.3
Claude Sonnet 4.6	103	12.6	13.6	−1.0
GPT-5.4-mini	226	0.0	8.4	−8.4

Table 7: Paired mechanism decomposition on problems where both PoT-retry's attempt-1 and RePoT's initial PoT failed. RePoT% is RePoT's conditional repair rate; Retry% the matched-budget fresh-sample rate; Δ = RePoT − Retry (pp).

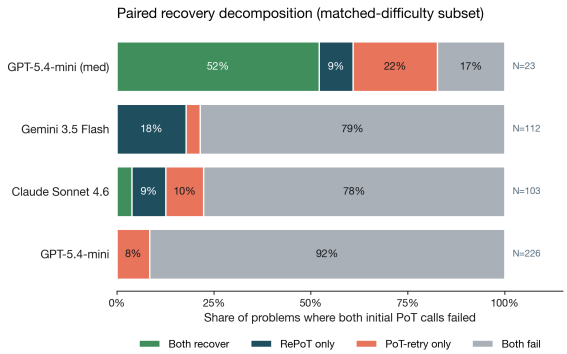

Figure 8: Paired recovery decomposition on the matched-difficulty subset (both methods' initial PoT failed). Stacks sum to 100% of N. "RePoT only" is mechanism evidence; "PoT-retry only" is fresh-sample evidence.

col) to defuse the "PoT baseline is weak" review concern. On 100 stratified problems with `gpt-5.4-mini-medium`: PoT final 89/100 vs PoT best-in-thought 90/100 (a single problem). RePoT is unaffected: 93/100 on both protocols. The +4pp RePoT − PoT delta is not an artifact of weak parsing.

G Cost decomposition appendix

We measured per-problem total token cost (prompt + completion) on `gpt-5.4-mini-medium` for the 100-problem stratified slice. The per-problem distribution decomposes into two clean modes:

- **No-repair runs (86% of problems).** RePoT's verified replay confirmed the initial PoT plan reaches the goal; cost is identical to PoT ($n = 86$, mean 1,820 tokens, median 1,540).
- **One-repair runs (14% of problems).** The initial PoT plan failed; RePoT issued one suffix-repair call. Per-problem cost is 1.4–1.7× PoT ($n = 14$, mean 2,610 tokens, median 2,180). The repair-call prompt is shorter than the initial PoT prompt (no examples, just the verified-prefix tail and the verifier error), but the model spends additional reasoning on resolving the failure boundary.

Aggregated over all 100 problems, RePoT averages 1.06× PoT cost. Per-problem cost data is released alongside the trace files.

Full per-method, per-model cost table. Aggregated over all 775 closed-model traces, Table 8 reports mean and median prompt+completion tokens, mean LLM calls, and mean wall-clock per problem for every (model, method) pair. RePoT and Adaptive RePoT average 1.11–1.39 LLM calls per problem, matching the PoT-retry budget.

Cost vs accuracy Pareto. Figure 9 visualizes the cost–accuracy trade-off mean-aggregated across the four closed-model configurations.

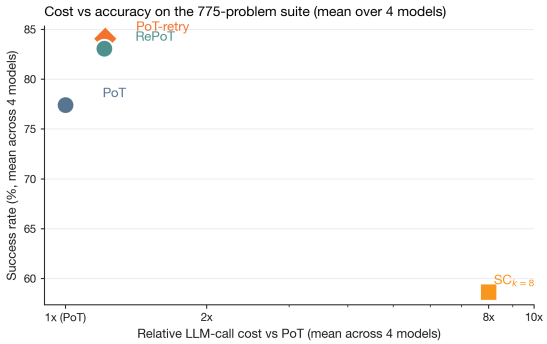

Figure 9: Cost vs accuracy, mean across the four closed-model configurations. PoT is the cost reference (1×). RePoT achieves PoT-retry-class accuracy at essentially the same mean cost (1.21×). $SC_{k=8}$'s low mean reflects its prose-plan format, which is more brittle than PoT's executable code.

Recovery decomposition. Figure 10 decomposes RePoT's wins into a second-attempt share (re-rolled PoT succeeds) and a mechanism share (suffix repair rescues a doubly-failed PoT).

H PlanBench per-complexity

Figure 11 shows the per-complexity breakdown for all three PlanBench models. The complete cell-level success rates (3 models × 10 complexities × 2 methods) are released as supplementary material. The two negative-delta cells (`gpt-5.4-mini` at $c = 8$ and $c = 11$, see Appendix J) appear in that file.

Model	Method	Mean tokens (in/out)	Median tokens (in/out)	Mean calls	Mean wall (s)
GPT-5.4-mini (med)	CoT	193 / 13323	184 / 8212	1.00	93.1
	$SC_{k=8}$	196 / 13277	184 / 7976	8.00	92.0
	PoT	216 / 4330	203 / 2294	1.00	29.9
	PoT-retry	245 / 4788	203 / 2377	1.14	41.4
	RePoT	257 / 4297	205 / 2166	1.11	34.9
Gemini 3.5 Flash	CoT	155 / 961	142 / 321	1.00	19.2
	$SC_{k=8}$	155 / 972	142 / 321	8.00	18.7
	PoT	210 / 497	197 / 510	1.00	6.5
	PoT-retry	251 / 598	197 / 636	1.18	9.4
	RePoT	272 / 484	199 / 418	1.17	7.6
Claude Sonnet 4.6	CoT	182 / 2185	168 / 1422	1.00	27.0
	$SC_{k=8}$	182 / 2374	168 / 1511	8.00	29.4
	PoT	236 / 891	222 / 743	1.00	12.9
	PoT-retry	274 / 1035	222 / 753	1.15	29.0
	RePoT	299 / 782	225 / 693	1.17	16.8
GPT-5.4-mini	CoT	197 / 1556	184 / 191	1.00	8.8
	$SC_{k=8}$	197 / 1209	184 / 191	8.00	6.9
	PoT	215 / 269	203 / 147	1.00	3.0
	PoT-retry	302 / 357	209 / 273	1.39	7.3
	RePoT	424 / 288	211 / 240	1.39	3.9

Table 8: Per-method cost breakdown across the three frontier models in four configurations, aggregated over all 775 problems. Tokens are summed across all LLM calls for the problem. Mean calls counts each LLM invocation (SC uses $k=8$; REPOT is $1+$ repair calls; POT-retry is 1 or 2). REPOT and ADAPTIVE REPOT average $\sim$1.1–1.4 LLM calls per problem, matching the POT-retry budget. Wall-clock is end-to-end per problem.

I Failure-mode case studies

We hand-analyze the five problems where POT succeeded but REPOT failed on the headline gpt-5.4-mini-medium run ($n = 100$ slice). All five share the same signature: REPOT's initial POT call returned an empty or non-goal-reaching plan, and the single repair call could not bridge the gap.

Taxonomy. PoT-call resampling loss (4/5): REPOT's initial POT call returned an empty or unparseable plan while standalone POT returned a goal-reaching plan — sampling variance, not a mechanism weakness. **Incomplete-plan stall (1/5):** the initial POT call emitted a 9-action plan that replayed cleanly but did not reach the goal; the repair call could not extend it within $T=4$ — a genuine single-call repair failure.

J Negative findings

We report two findings that complicate the headline narrative. We include them rather than buried them.

(i) DERAIL prefix-flip. On both Gemini and GPT (med), REPOT $_{\text{restart}}$ beats REPOT $_{\text{full}}$ (82.4 vs 80.7 on Gemini; 94.5 vs 59.6 on GPT (med) — a much larger gap than on Gemini). The inequality REPOT $_{\text{full}}$ >REPOT $_{\text{no-prefix}}$ does hold on both (+3.8 / +5.8pp), so the prefix tail is not actively harmful, but anchoring the repair on the post-injection state appears to confuse the model in this controlled-error setting. The gap to non-checkpointed baselines ($\geq$ 30pp on GPT (med), $\geq$60pp on Gemini) remains the load-bearing finding. We hypothesise that the injected mid-rollout state is sometimes unreachable from the goal under a single repair call, while a fresh plan from s_0 avoids that trap. Reporting as-is.

(ii) PlanBench gpt-5.4-mini at $c=8$ and $c=11$. At these two complexities, REPOT underperforms POT by -8.3pp and -9.1pp respectively. Both points fall in the regime where POT has very low base accuracy (33% and 18%) and the mid-rollout state REPOT resumes from is itself misleading. The single repair call sometimes commits to extending a bad prefix. A higher repair budget, or a verifier-driven decision to restart from s_0 when the prefix-fraction is low, would mitigate; we treat both as future work and report the negative cells transparently.

K Algorithm: full code listings and repair prompt

The REPOT agent's `run()` method, the `replay_until_failure()` primitive, and the `code_prompt()` repair template are released

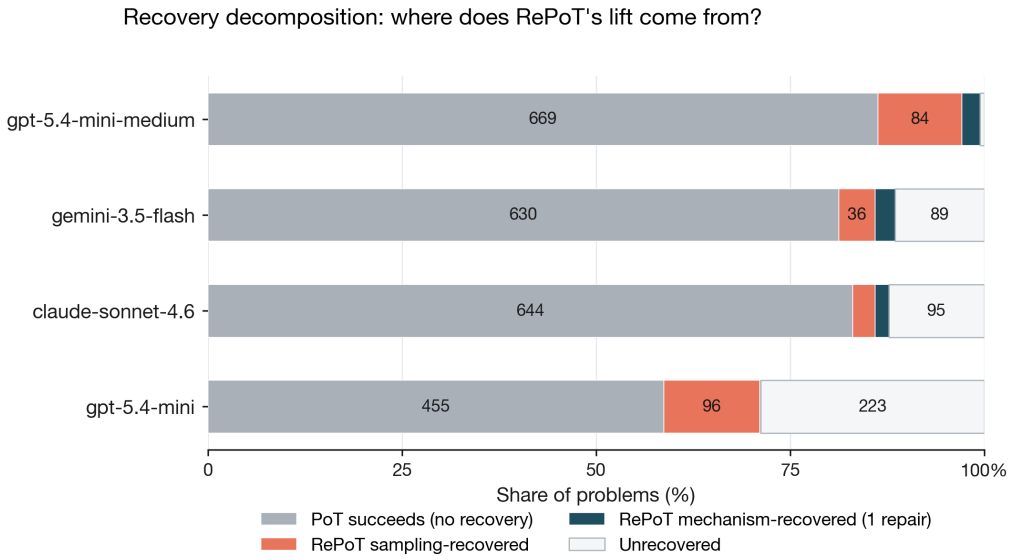

Figure 10: Decomposition of REPOT's recovery on PUZZLEZOO-775. Each row is one model; bars sum to 100%. The **teal** segment is the share of problems where standalone POT fails, REPOT's first POT call also fails, and the suffix repair call rescues (*mechanism*). The **coral** segment is the second-attempt contribution (standalone POT fails but REPOT's re-rolled POT call succeeds without invoking the repair). On the strongest model most of REPOT's lift is second-attempt; the verified-prefix repair contributes the harder tail. The matched-budget POT-retry comparison in Table 1 isolates the mechanism contribution under equal LLM-call budget.

alongside the paper. The full implementation is approximately ∼25 lines for `run()`, ∼25 for the replay primitive, and ∼60 for the prompt; see the released source for verbatim listings. Figure 12 below shows the verified-prefix-conditioned repair prompt template.

L Ablation conditions

The three named ablations referenced in §4.5 and used in §7:

REPOT $_{\text{full}}$ the algorithm of Algorithm 1.

REPOT $_{\text{no-prefix}}$ same as full, but the dynamic-block prefix-tail and "`|P| verified moves`" counter are hidden; the model sees only the current verified state and the error message. Tests whether the prefix tail is what the model uses, not just the checkpoint state.

REPOT $_{\text{restart}}$ same budget, but the repair call restarts from s_0 instead of the verified state s. Tests whether the gain is the verified checkpoint or just an extra LLM call.

M DERAIL condition definitions

DERAIL measures recovery from one injected wrong action at a ∼1/3 checkpoint of the oracle plan. We compare 11 conditions:

`no_feedback`. The model sees the post-injection state only; no error message, no checkpoint marker.

`error_only`. The model sees the post-injection state and a one-line verifier error; no checkpoint visualization.

`state_feedback`. The model sees the last valid checkpoint state plus the wrong action that was attempted.

`state_plus_legal_actions`. As above, plus the legal action set from the checkpoint.

`stateguard_rollback`. A controller-based rollback baseline: the controller proposes one action at a time, the verifier accepts or rejects, repeat until budget exhaustion.

REPOT variants. Three REPOT conditions (`repot_full`, `repot_no_prefix`, `repot_restart`) matching App. L.

All conditions share the same per-problem injection seed; comparisons are paired across conditions on the same problem.

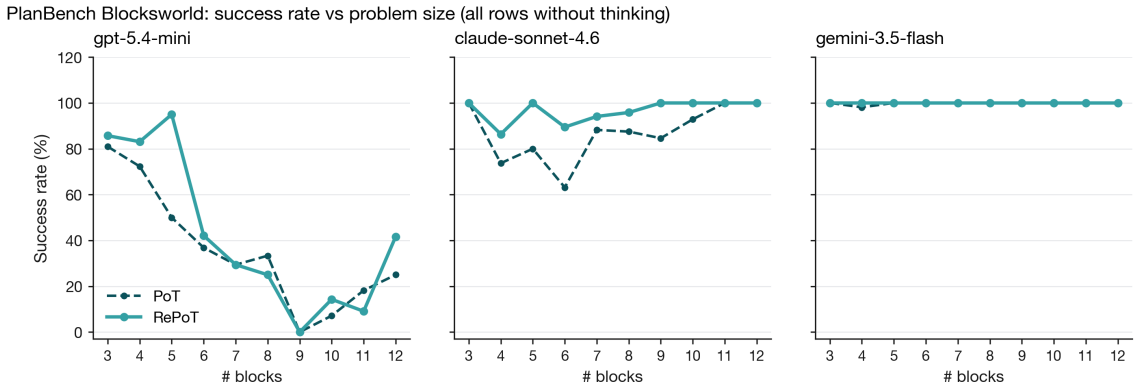

Figure 11: Per-complexity success rate on PLANBENCH BLOCKSWORLD. REPOT's lift concentrates in the mid-complexity band where POT has both failures to recover from and enough valid prefix to recover into. Two negative-delta cells (gpt-5.4-mini at $c = 8, 11$) are reported in Appendix J.

Stable block (cacheable):

```
{problem.natural_language_prompt}
Goal state: {goal_state}
Write Python code that prints exactly one line:
moves = [...]
containing up to K primitive moves to apply
from the current verified state.
— verifier checkpoint below —
```

Dynamic block:

```
You have already executed |P| verified moves.
Recent verified moves: {tail_T(P)}
Current verified state: {s}
Legal moves: {legal_actions(s)}
Blocked: {blocked(s)}
Verifier message: {ϵ}
```

Figure 12: Verified-prefix-conditioned repair prompt. The "`verifier checkpoint`" marker delimits the cacheable problem description from the per-call dynamic state.

N Open-source results: routing, cost, and capability scaling

This appendix expands on §6.4 with the adaptive policy routing breakdown, per-model cost statistics, and the capability-scaling regression. Per-model success rates for the five core methods are in the bottom block of Table 1; the ADAPTIVE REPOT success rates are 60.0% (Gemma 4 26B-A4B), 64.2% (GPT-OSS 20B), 62.5% (Qwen 3.6 35B-A3B), and 13.3% (Nemotron-3 Nano 30B FP8) on the same 120-problem stratified subset.

Adaptive policy routing. Figure 13 shows the routing distribution of ADAPTIVE REPOT across the four open-source models. On the three workable models (Gemma, GPT-OSS, Qwen) about half of problems trigger the *initial PoT success* branch and most of the remainder route to *fresh retry (empty plan)*: the model emits an empty or near-empty plan when it cannot solve, and the suffix-repair branch activates on only 1–3 problems out of 120. This is a different mechanism profile than closed thinking-on models, where suffix repair is the dominant value channel. Nemotron-3 Nano 30B FP8 routes mostly to short-prefix retries with a low rescue rate (3%), confirming the capability-floor reading.

Open-source cost. Mean total tokens per problem (prompt + completion) under REPOT are 1.5–2× POT on Gemma, GPT-OSS, and Qwen, in line with the 1.06–1.4× closed-model range. Wall-clock per problem: POT 25–60 s, REPOT 40–110 s. Detailed per-method, per-model numbers are released alongside the trace files.

Capability scaling regression. The REPOT −POT-retry success-rate delta regressed on the mean verified-prefix fraction of failed initial POT plans across all (model, environment) cells appears in Fig. 3 (§6.4).

O Release

We will release the source code (REPOT, ADAPTIVE REPOT, the verified-replay primitive, the four environments, the DERAIL harness, and the prompt templates), PUZZLEZOO-775 and DERAIL-550 as data, and all closed-model and open-source trace files under a permissive license. The review version is available at an anonymized repository (link withheld).

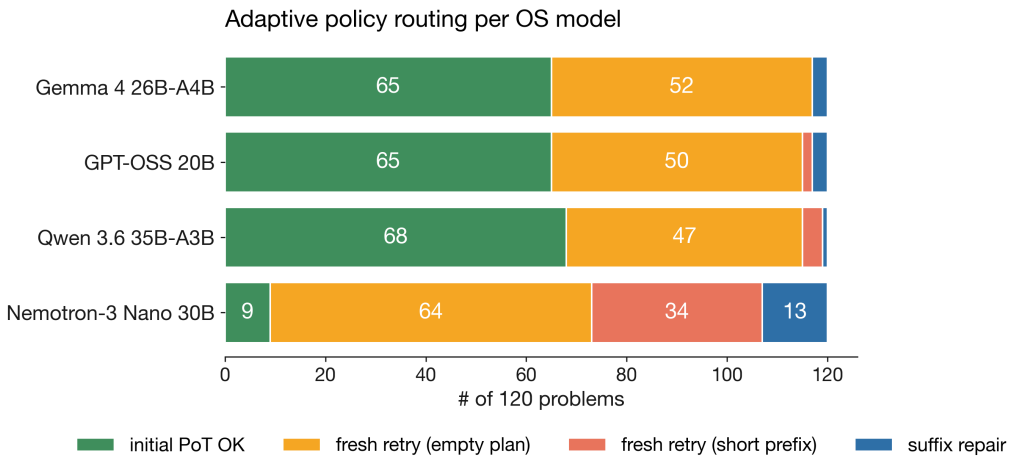

Figure 13: Adaptive policy routing per open-source model. Bars sum to 120 problems. The three workable rows are dominated by *initial PoT success* + *fresh retry (empty plan)*; suffix repair is rare. Nemotron-3 (bottom) shows a markedly different shape: few initial successes, many short-prefix retries.