REPO: REPLAY-ENABLED POLICY OPTIMIZATION FOR TERMINAL AGENTS VIA TRAJECTORY-RECONSTRUCTED ENVIRONMENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Realistic, executable environments are scarce for terminal-agent post-training, while recorded trajectories are plentiful but remain fixed demonstrations. We introduce **REPO** (Replay-Enabled Policy Optimization), which reconstructs an executable workspace from each trajectory by deterministic replay and agentic completion, then adds per-turn container snapshots for counterfactual rollouts. The recovered workspace supports intent recovery, new single- and cross-workspace tasks, and multi-round sessions, each paired with an executable verifier. It expands each workspace along breadth (cross-project dependencies) and depth (persistent user follow-ups). We further introduce PAPO, which anchors sibling rollouts at verifier-diagnosed pivotal turns, and CBAO, which branches at high-entropy turns to obtain local credit signals. On public trajectories, REPO yields 37.3k task-sufficient environments. SFT warm-start followed by PAPO+CBAO improves Qwen3.5-27B by 11.9 points on Terminal-Bench 2.1 and by 18.3 points in EvoCode-Bench v2 MT@4.

1 INTRODUCTION

Terminal-agent trajectories are abundant, but realistic executable environments remain scarce. A trajectory is a fixed demonstration whose outcome and workspace state cannot be re-tested; an environment can be re-solved by stronger policies, verified, and queried repeatedly. Expert benchmarks such as Terminal-Bench (Merrill et al., 2026) provide this reliability through manually built containers and tests, but manual curation limits scale. REPO addresses this mismatch by turning trajectories into reusable, verifiable environments.

Existing work scales environments through repository history (Pan et al., 2024; Jain et al., 2025), perturbation of functioning projects (Yang et al., 2025; Lin et al., 2026), or task-conditioned synthesis from categories and skill structures (Gandhi et al., 2026; Ivison et al., 2026; Hua et al., 2026; Fan et al., 2026). These routes either concentrate on repair tasks or generate environments detached from the state of a real agent interaction.

These differences matter for RL post-training. Repository rollback preserves a real codebase but usually exposes a single historical issue and its associated test; perturbation provides diversity but remains tied to injected defects; and task-conditioned synthesis offers broad control but must ask a generator to invent both project context and executable checks. None of these constructions directly preserves the sequence of intermediate states that a terminal agent actually visited. Consequently, a failed long-horizon rollout still supplies only a terminal failure signal, making it difficult to tell whether the failure was caused by one pivotal edit or by an inability to continue after a correct edit. This distinction is essential when allocating expensive RL rollouts.

Trajectories are observations of such environments and already expose their contents through reads, writes, edits, and tool outputs. REPO uses this trace to reconstruct a replay-enabled environment with per-turn snapshots, then trains a policy on it with replay-native RL after an SFT warm-start (Figure 1).

The reconstruction is deliberately conservative: replay restores the earliest observed version of accessed files and holds out the agent's later edits, while an agentic completion step supplies missing

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.

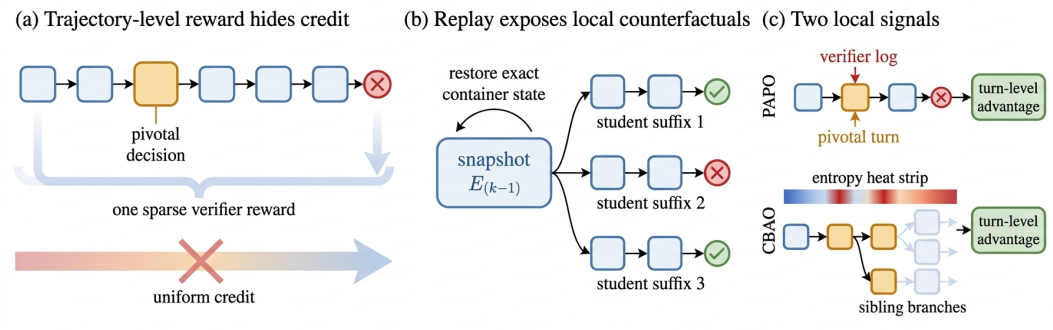

Figure 1: **Why replay-native credit assignment is needed.** A terminal verifier emits one sparse reward after a long action chain, hiding the pivotal decision. Replaying the exact container state enables local sibling rollouts, while PAPO and CBAO use verifier-diagnosed and high-entropy turns to produce turn-level advantages.

project context without solving the target task. The resulting container, rather than a learned simulator, is the environment. A commit after every turn makes the pre-action state recoverable, so sibling suffixes can be evaluated under the same files, processes, and dependencies. This gives REPO a concrete counterfactual unit for credit assignment and avoids treating a teacher's complete solution as uniformly useful supervision.

REPO builds its training substrate around a resettable execution state rather than around a finished demonstration. The replay stage records the earliest task-relevant evidence for every accessed path and withholds edits produced after the corresponding action. A completion model then repairs the missing project context under an explicit constraint: it may make the task executable, but it may not implement the requested change. We commit the resulting container after each interaction turn, so PAPO can revisit the state preceding a verifier failure and CBAO can branch from the same state when the policy is uncertain (§ 3.1).

The reconstructed state also supports four data-generation modes. Intent Recovery preserves the user-stated objective, whereas Single-WS and Cross-WS create new requirements within one workspace or across a dependency pair. Multi-Round extends a solved state with subsequent user decisions and verifier feedback. Cross-WS therefore tests whether a policy can transfer a capability across codebases, while Multi-Round tests whether it can preserve and revise a contract over time (§ 3.1). An in-container verifier checks each generated task; only teacher trajectories that satisfy the resulting executable contract enter the training corpus (§ 4.1).

Applied to public terminal-agent traces, this procedure yields 37.3k task-sufficient environments. We use Qwen3.7-Max to generate solution rollouts and train Qwen3.5-27B with an SFT warm start followed by the replay-native objectives. In the reported evaluation, the resulting model improves Terminal-Bench 2.1 by 11.9 points and EvoCode-Bench v2 MT@4 by 18.3 points (§ 4). Controlled comparisons then separate the gain from reconstructing executable state, expanding the task distribution, and assigning credit at verifier-identified or high-entropy turns.

To sum up, we make the following contributions:

- **Replay-native RL substrate.** We turn observed tool traces into resettable containers with a per-turn state chain, allowing the policy to revisit the execution context in which a decision was made while keeping the teacher's later edits hidden.
- **Local credit assignment.** PAPO uses verifier evidence to anchor sibling rollouts at likely failure turns, and CBAO allocates contrastive branches to high-entropy turns. Both mechanisms operate on the same restored states and are combined under a KL-regularized Stage-2 objective.
- **Task expansion and evaluation.** We retain the original intent-recovery setting while adding single- and cross-workspace requirements and persistent user follow-ups. On the resulting corpus, Qwen3.5-27B gains 11.9 points on Terminal-Bench 2.1 and 18.3 points in EvoCode-Bench v2 MT@4; controlled ablations isolate the contributions of reconstruction, task breadth, session depth, and local credit assignment.

Table 12 is provided in the appendix as a compact reference for the environment-construction landscape.

2 Related Work

Environment construction for terminal agents. Prior approaches obtain executable workspaces in three ways. Repository-based methods recover states tied to historical issues or commits, as in SWE-Gym and R2E-Gym (Pan et al., 2024; Jain et al., 2025). Perturbation methods begin with functioning repositories or CLI workspaces and inject bugs or failures, as in SWE-smith and CLI-Gym (Yang et al., 2025; Lin et al., 2026). SETA scales verifiable environments through synthesis and adaptive evolution (Shen et al., 2026b). REPO follows a different starting point: it replays the file operations in recorded agent trajectories, completes the missing project context, and turns the resulting workspaces into executable RL environments for PAPO and CBAO.

Task synthesis and environment grounding. Task-conditioned pipelines generate task–environment pairs from categories, seeds, capability taxonomies, compositional axes, or skill graphs (Gandhi et al., 2026; Ivison et al., 2026; Hua et al., 2026; Fan et al., 2026; Zhu et al., 2026). Other systems use agent skills, executable meta-tasks, or solver feedback to diversify and calibrate generated tasks (Cheng et al., 2026; Pan et al., 2026; Meng et al., 2026). Environment-grounded methods instead perturb healthy code, mine documentation, use real issues, or jointly construct instructions, solutions, and verifiers (Lin et al., 2026; Wu et al., 2026a; Yang et al., 2026; Shi et al., 2026). RST recursively expands verified seeds while realigning the associated task, verifier, and environment (Li et al., 2026b). These methods construct useful tasks, but REPO additionally preserves the intermediate workspace states exposed by an actual agent trajectory and uses them for replay-native RL.

Interactive and multi-round agents. Interactive benchmarks study coding as a sequence of requests rather than an isolated prompt. InterCode provides execution feedback through a containerized shell (Yang et al., 2023); SWE-INTERACT and SWE-Together model progressive user feedback and revisions (Raghavendra et al., 2026; Wu et al., 2026b); and ICAE-Bench evaluates project construction from incomplete requirements with an automated user (Peng et al., 2026b). EvoCode-Bench v2 evaluates persistent workspaces across sequential development requests (Shen et al., 2026a). REPO adopts this setting as its depth axis, extending reconstructed workspaces into verified multi-round sessions, while its breadth axis creates cross-workspace tasks spanning dependent codebases.

Table 12 summarizes the distinction: REPO uses trajectories as the source of replayable environments, expands them both across workspaces and over user rounds, and trains with snapshot-based RL objectives rather than relying on SFT alone.

3 REPO

Figure 2 presents the pipeline. We first cast REPO as an executable RL environment (§ 3.1), specify the two-stage training scheme in which SFT warm-starts a reference policy for RL post-training (§ 3.2), and then describe the two RL algorithms that constitute the paper's methodological contribution: pivotal-anchored policy optimization (§ 3.3) and contrastive branch advantage optimization (§ 3.4). A unified objective and cost analysis close the section (§ 3.5).

3.1 REPO as an RL Environment

States, actions, and transitions. Each task defines a finite-horizon MDP $(\mathcal{S}, \mathcal{A}, T, R, \gamma)$. A state $s_t = (\widehat{E}_t, h_t)$ pairs the current workspace $\widehat{E}_t$ (files, working directory, and process state inside a standardized `ubuntu:24.04` container) with the interaction history h_t. An action a_t is a full agent turn drawn from $\pi_\theta(a_t \mid s_t)$: a token sequence containing internal reasoning and one or more tool calls (shell commands, file edits, submissions). The transition $s_{t+1} = T(s_t, a_t)$ is deterministic and enacted by executing a_t's tool calls inside the live container; the environment therefore *is* the container, not a learned model of it.

Environment construction from trajectories. We recover the workspace $\widehat{E}$ from an existing agent trajectory τ in three stages: (i) *deterministic replay* restores every accessed path to its earliest observed version, yielding a partial workspace $\widehat{E}_0$ with the agent's own outputs held out as verification

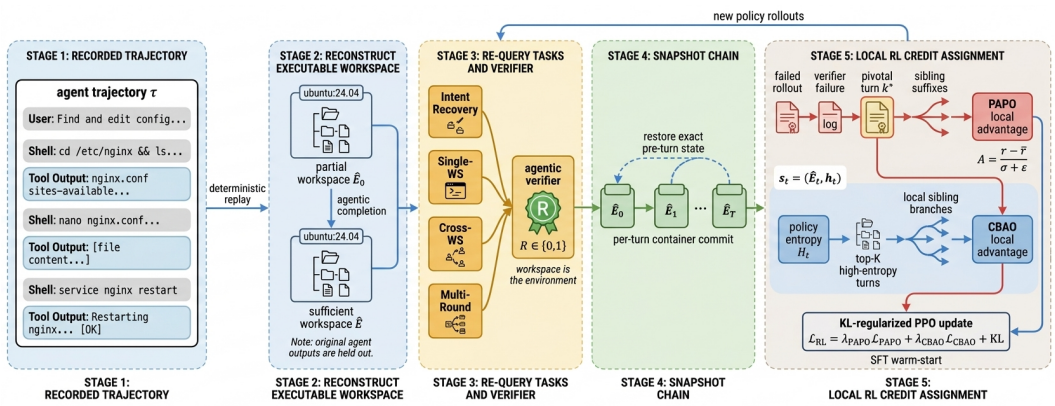

Figure 2: **Framework of REPO.** Recorded trajectories are replayed and completed into executable workspaces, expanded into four verified task families, and retained with a per-turn snapshot chain. Restored snapshots support PAPO on verifier-identified pivotal turns and CBAO on high-entropy turns; their local advantages drive the KL-regularized RL update.

ground truth; (ii) *agentic completion* supplies missing files and dependencies to make the recovered task q solvable without implementing it; (iii) *sufficiency filtering* discards workspaces on which an agentic judge finds insufficient project context (§ C.3). Full rules and prompts are in §§ C.1 and C.2; standardized containers avoid per-repository image cost at a modest resolve-rate penalty (Zeng et al., 2026).

Tasks and verifier-supplied reward. Each reconstructed workspace is turned into one or more RL tasks by four complementary re-querying variants (full prompts in § D): *Intent Recovery* consolidates the source trajectory's user requests into a single task; *Single-WS* synthesizes new within-workspace tasks under groundedness, diversity, and verifiability constraints; *Cross-WS* pairs a writable target workspace with a read-only reference workspace it functionally depends on, and asks the solver to bridge the gap without seeing the reference's internals; *Multi-Round* preserves the workspace across coding rounds and lets a user agent issue grounded follow-ups. Every task is paired with an agentic verifier that authors a self-contained `pytest` suite inside the container (§ E). The verifier defines the reward function: for single-round tasks, $R(\tau) = \mathbb{1}[\text{all tests pass}] \in \{0, 1\}$ is emitted at the terminal state; for Multi-Round, per-round rewards $r_i \in \{0, 1\}$ arrive at each round boundary, and we treat the session as a factored MDP over rounds. Rewards are otherwise sparse, so credit assignment across long horizons is the central RL challenge.

Replayable snapshot chain. During any rollout we take a lightweight incremental container commit after each turn, yielding a chain of snapshots $\{\widehat{E}_0, \ldots, \widehat{E}_{T-1}\}$. Restoring $\widehat{E}_{k-1}$ recovers the exact pre-turn state so that any turn k can be replayed from a different action a'_k under π_θ. This property, inherited directly from the trajectory-to-environment reconstruction of the previous paragraph, is the structural asset that both of our RL algorithms exploit: it turns credit assignment from a purely trajectory-level problem into one we can attack at the turn level with concrete counterfactual rollouts. No other environment-synthesis framework we are aware of provides this, because their environments are not themselves built by replay.

3.2 Two-Stage Training: SFT Warm-Start and RL Post-Training

RL from scratch is infeasible in this setting: with a verifier reward pass rate of 49.2% on Cross-WS tasks under a strong teacher (Table 15), a randomly initialized policy would receive near-zero reward and its policy gradient would collapse. We therefore adopt a two-stage recipe. In *Stage 1 (SFT)*, verifier-accepted teacher trajectories over the reconstructed environments are used to train an initial policy π_{SFT} by standard maximum likelihood, following the corpus in § 4.1. In *Stage 2 (RL)*, we initialize $\pi_\theta \leftarrow \pi_{\text{SFT}}$ and continue training on the same environments with the algorithms of §§ 3.3 and 3.4. π_{SFT} then plays three coupled roles in Stage 2: it is the initialization of π_θ, the default sampling distribution for on-policy rollouts (so the initial rollout pass rate is close to the teacher's, not zero), and the anchor of a KL regularizer that prevents the policy from drifting into behavior no longer

solvable by the verifier. Concretely we regularize with both a step-wise trust region $\mathrm{KL}(\pi_\theta \| \pi_{\mathrm{old}})$ (with π_{old} the policy that generated the current batch) and a long-horizon anchor $\mathrm{KL}(\pi_\theta \| \pi_{\mathrm{SFT}})$, which is standard in RLHF and, in our sparse-reward regime, we found necessary for stability.

3.3 PIVOTAL-ANCHORED POLICY OPTIMIZATION (PAPO)

Motivation. Vanilla GRPO-style group rollouts sample G independent trajectories from $t=0$ per prompt and compute a trajectory-level baseline. Under our verifier, most groups on hard tasks collapse to all-fail, and the informative variance sits at a small number of *pivotal* turns where a single action decides the outcome. PAPO relocates the group baseline to those turns.

Pivotal turn identification. For a verifier-failed rollout $\tilde{\tau}$ we form a candidate set anchored in verifier-visible state. Let $\mathcal{F}_{\mathrm{fail}}(\tilde{\tau})$ be the paths, commands, and stack frames named in the failing tests' log. Define

$$\mathcal{U}(\tilde{\tau}) \;=\; \big\{\, t \in [T] \;:\; \tilde{\tau}_t \text{ writes, edits, or executes an element of } \mathcal{F}_{\mathrm{fail}}(\tilde{\tau}) \,\big\}.$$

A single teacher call selects $k^\star \in \mathcal{U}(\tilde{\tau})$ (the earliest turn responsible for failure) together with a one-line correction hint remark. Restricting $k^\star$ to $\mathcal{U}$ grounds the pivot in the verifier's own failure signature and avoids the drift of self-report reflection.

Group rollout from the pivotal snapshot. We restore $\widehat{E}_{k^\star-1}$, expose $\tilde{\tau}_{<k^\star}$ and remark as history, and draw G sibling suffixes $\{\tau^{(g)}_{\geq k^\star}\}_{g=1}^{G} \sim \pi_{\mathrm{old}}(\cdot \mid s_{k^\star-1})$ under the same decoding configuration used at evaluation. Each is executed against the verifier, yielding rewards $r^{(g)} \in \{0,1\}$. This is on-policy sampling starting from a state that lies deep inside the horizon.

Pivotal-anchored advantage. We compute a group-relative advantage *local to the pivotal state*:

$$\hat{A}^{\mathrm{piv}}_g \;=\; \frac{r^{(g)} - \bar{r}_{k^\star}}{\sigma_{k^\star} + \epsilon}, \qquad \bar{r}_{k^\star} = \tfrac{1}{G}\sum_g r^{(g)}, \;\; \sigma_{k^\star} = \sqrt{\tfrac{1}{G}\sum_g (r^{(g)} - \bar{r}_{k^\star})^2}.$$

Because $\tilde{\tau}_{<k^\star}$ has already been implicitly endorsed by our choice of $k^\star$ as the *earliest* error, we do not update the policy on the prefix; the advantage is broadcast only to the tokens generated at turns $t \geq k^\star$. This "pivotal isolation" is the mechanism by which credit is confined to the causal turns and no gradient signal is placed on tokens whose correctness is not in question.

Clipped surrogate. With $\rho_{g,t}(\theta) = \pi_\theta(a_t \mid s_t)/\pi_{\mathrm{old}}(a_t \mid s_t)$, the PAPO objective is

$$\mathcal{L}_{\mathrm{PAPO}}(\theta) \;=\; -\,\mathbb{E}_{g,\,t\geq k^\star}\Big[\min\big(\rho_{g,t}(\theta)\,\hat{A}^{\mathrm{piv}}_g,\; \mathrm{clip}(\rho_{g,t}(\theta),\,1-\varepsilon,\,1+\varepsilon)\,\hat{A}^{\mathrm{piv}}_g\big)\Big].$$

This is a standard PPO/GRPO surrogate, but its expectation is taken over sibling rollouts from the same pivotal state rather than from $t=0$; the importance ratio is well-defined because both π_θ and π_{old} share the same restored context. PAPO is applied to Cross-WS and Multi-Round rollouts, where teacher pass@1 is lowest and the marginal informativeness of a pivotal group is largest.

Algorithm 1 PAPO update on one failed rollout

1: **Input:** failed rollout $\tilde{\tau}$, snapshot chain $\{\widehat{E}_{t-1}\}$, failing-test log ℓ, group size G, clip ε.
2: $\mathcal{U} \leftarrow \{t : \tilde{\tau}_t \text{ touches paths/commands in } \ell\}$; $(k^\star, \mathrm{remark}) \leftarrow \textsc{TeacherReflect}(\tilde{\tau}, \mathcal{U}, \ell)$
3: Restore $\widehat{E}_{k^\star-1}$; sample $\{\tau^{(g)}_{\geq k^\star}\}_{g=1}^{G} \sim \pi_{\mathrm{old}}$; obtain rewards $\{r^{(g)}\}$ from the verifier.
4: Compute $\hat{A}^{\mathrm{piv}}_g$ via group-relative normalization; broadcast to tokens at $t \geq k^\star$.
5: Update θ by one clipped-surrogate step on $\mathcal{L}_{\mathrm{PAPO}}$; skip the batch if $\sigma_{k^\star} < \epsilon$.

3.4 CONTRASTIVE BRANCH ADVANTAGE OPTIMIZATION (CBAO)

Motivation. PAPO targets failed rollouts, but passing rollouts also traverse decision-sensitive turns whose signal a trajectory-level baseline dilutes. CBAO turns every rollout, passing or failing, into a set of *local* group advantages anchored at the turns where the policy is most uncertain, providing a fine-grained credit signal that composes with PAPO on the same snapshot chain.

Branch candidate turns. For any rollout τ we compute the policy's action-token entropy H_t at each turn from the same logprobs used during generation (thought tokens masked so the signal reflects uncertainty over what to *do*, not what to *say*). The top-K turns by H_t form the branch set $\mathcal{B}(\tau)$; branching only at high-entropy turns concentrates the sampling budget where policy improvement is possible.

Local group rollout and branch reward. For each $k \in \mathcal{B}(\tau)$ we restore $\widehat{E}_{k-1}$, expose $\tau_{<k}$ as history, and draw B sibling suffixes from π_{old}; combined with the original suffix this yields $B+1$ verifier rewards $\{r_k^{(b)}\}_{b=0}^{B}$ with $r_k^{(0)}$ the reward of τ itself.

Contrastive branch advantage. The branch-local group-relative advantage is

$$\hat{A}_{b,k}^{\text{cba}} \;=\; \frac{r_k^{(b)} - \bar{r}_k}{\sigma_k + \epsilon}, \qquad \sigma_k = \sqrt{\tfrac{1}{B+1}\sum_b (r_k^{(b)} - \bar{r}_k)^2}.$$

Here σ_k is the local variance of verifier outcomes around τ's k-th turn. When σ_k is large (*decision-sensitive* turn where siblings split between pass and fail), the advantage takes large signed values and drives a strong update; when $\sigma_k \to 0$ (turn is trivially solvable or hopeless), the ϵ in the denominator drives the advantage to zero and the branch contributes no gradient. The variance is thus not just a filter but the normalizer of the advantage itself, which is the mechanism by which CBAO automatically routes gradient to the informative turns.

Clipped surrogate. With $\rho_{b,t}(\theta) = \pi_\theta(a_t \,|\, s_t)/\pi_{\text{old}}(a_t \,|\, s_t)$, the CBAO objective sums the per-branch clipped surrogate over the sampled branches:

$$\mathcal{L}_{\text{CBAO}}(\theta) \;=\; -\,\mathbb{E}_{k \in \mathcal{B}(\tau),\, b,\, t \geq k}\Big[\min\big(\rho_{b,t}(\theta)\,\hat{A}_{b,k}^{\text{cba}},\; \text{clip}(\rho_{b,t}(\theta),\, 1-\varepsilon,\, 1+\varepsilon)\,\hat{A}_{b,k}^{\text{cba}}\big)\Big].$$

Whereas GRPO computes one advantage per trajectory, CBAO computes $K \cdot (B+1)$ advantages per trajectory, each anchored to a distinct high-entropy state, and clips them independently. It is the local, high-resolution counterpart to GRPO's trajectory-level baseline.

Algorithm 2 CBAO update on one rollout

1: **Input:** rollout τ, snapshot chain $\{\widehat{E}_{t-1}\}$, action-token entropies $\{H_t\}$, top-K, branch width B, clip ε.
2: $\mathcal{B}(\tau) \leftarrow$ top-K turns by H_t
3: **for** $k \in \mathcal{B}(\tau)$ **do**
4: Restore $\widehat{E}_{k-1}$; sample $\{\tau_{\geq k}^{(b)}\}_{b=1}^{B} \sim \pi_{\text{old}}$; obtain rewards $\{r_k^{(b)}\}_{b=0}^{B}$.
5: Compute $\hat{A}_{b,k}^{\text{cba}}$ via local group-relative normalization; skip the branch if $\sigma_k < \epsilon$.
6: **end for**
7: Take one PPO-style optimizer step on $\mathcal{L}_{\text{CBAO}}$ using all $\hat{A}_{b,k}^{\text{cba}}$ pooled across branches.

3.5 Unified Objective, Cost, and Ablative Reductions

Unified objective. Stage-2 RL optimizes the composite

$$\mathcal{L}_{\text{RL}}(\theta) \;=\; \lambda_{\text{PAPO}}\,\mathcal{L}_{\text{PAPO}}(\theta) \;+\; \lambda_{\text{CBAO}}\,\mathcal{L}_{\text{CBAO}}(\theta) \;+\; \beta_{\text{old}}\,\text{KL}\big(\pi_\theta \| \pi_{\text{old}}\big) \;+\; \beta_{\text{SFT}}\,\text{KL}\big(\pi_\theta \| \pi_{\text{SFT}}\big),$$

where the two KL terms serve distinct purposes: the step-wise $\text{KL}(\pi_\theta \| \pi_{\text{old}})$ enforces a trust region on each policy update, while the long-horizon $\text{KL}(\pi_\theta \| \pi_{\text{SFT}})$ prevents cumulative drift away from the SFT initialization. Both are standard in RLHF and, in our sparse-reward setting, we found the anchor term necessary for stability across long training runs. Setting $\lambda_{\text{PAPO}} = 0$ or $\lambda_{\text{CBAO}} = 0$ recovers each algorithm in isolation, so the two contributions are directly ablatable.

Cost and snapshot sharing. Both algorithms consume the same per-turn snapshot chain built during the initial on-policy rollout: PAPO uses one restored snapshot per failed rollout for its G sibling suffixes; CBAO uses K restored snapshots per rollout for its branch groups. Because sibling suffixes start from a mid-episode snapshot rather than from $t = 0$, the expected token cost of a sibling is a small fraction of a full rollout. The total Stage-2 overhead per rollout is therefore bounded by $G + K(B+1)$ partial rollouts on the shared snapshot chain, independently of which algorithm fires.

4 EXPERIMENTS

4.1 SETUP

Data. We source raw agent trajectories from diverse terminal-style CLI and software-engineering corpora (§ B); a trace is retained as a seed when its terminal workspace contains at least 5 files and 100 lines, and Terminal-Bench-derived sources are removed before reconstruction. The pipeline yields 68,263 candidate workspaces and 37,273 task-sufficient environments after completion and deduplication. The reconstruction census and language/domain breakdown are reported in Appendix § H. Across the retained environments, verifier-filtered Single-WS, Cross-WS, and Multi-Round synthesis yields 31,977 SFT demonstrations (25,386 Single-WS, 3,512 Cross-WS, 3,079 Multi-Round) totaling approximately 1.42B tokens (median interaction statistics in § I).

Training. We first fine-tune Qwen3.5-27B by SFT on the verifier-accepted REPO trajectories for two epochs to obtain the reference policy π_{SFT}. We then run Stage-2 RL with the unified PAPO+CBAO objective in § 3.5; PAPO is applied to verifier-failed rollouts and CBAO to high-entropy turns from both passing and failing rollouts. Unless otherwise stated, the reported REPO model is the joint PAPO+CBAO model, while SFT, PAPO-only, and CBAO-only are controlled ablations with the same data and optimizer budget. We use a constant learning rate of 7×10^{-6}, a global batch size of 256, a 256k-token sequence length, $G = 4$ pivotal siblings, $K = 3$ entropy-selected branches, and $B = 3$ contrastive siblings. Before training, we run a 13-gram contamination check against Terminal-Bench tasks and exclude Terminal-Bench-derived source datasets from all four re-querying variants.

Single-round evaluation (Terminal-Bench 2.0 and 2.1). We evaluate Terminal-Bench with Claude Code (version 2.1.126) and Terminus2 (Merrill et al., 2026) using the XML parser. Evaluation mirrors the decoding and execution configurations of the solution rollout (§ 3.1), employing temperature 1.0, top-$p = 0.95$, interleaved thinking, a 256k context window, a 65,536-token turn limit, proactive summarization at 176k tokens, up to 500 agent turns, and a four-hour wall-clock timeout. For Claude Code, interactive and web-retrieval tools (`WebFetch`, `WebSearch`, `AskUserQuestion`, `EnterPlanMode`, `ExitPlanMode`) are disabled. Each container runs with 12 CPU cores and 32 GiB of memory. Reported scores represent mean pass rates over six independent runs. Differences between Terminal-Bench 2.0 and 2.1 are documented in the benchmark update.[1]

Multi-round evaluation (EvoCode-Bench v2). We evaluate depth expansion on EvoCode-Bench v2[2] (Shen et al., 2026a), which contains 26 coding tasks and 227 rounds (5–15 requests per task). Workspaces and sessions persist, and cumulative verifiers check both current and active prior requirements. Our Qwen3.5-27B checkpoints use Terminus2-XML at temperature 1.0 with interleaved thinking, a 256k-token context, a per-turn limit of 65,536 tokens, proactive summarization at 176k tokens, up to 500 agent turns per request round, and a total wall-clock limit of 10 hours per stateful task. Each task is evaluated in four independent runs. MT@4 uses fail-stop scoring: a run receives credit only for the consecutive rounds it passes before its first failure, and each task-round is credited if any run reaches it successfully. The score averages credited rounds within each task and then across tasks. Case score averages verifier-case pass rates within tasks and then across tasks and runs to measure partial progress.

4.2 MAIN RESULTS

Table 1 reports single- and multi-round performance. The SFT warm-start reaches 53.8% on Terminal-Bench 2.1 under Terminus2-XML. Applying PAPO+CBAO on the same reconstructed environments raises this to 58.1% (+11.9 over the base model), with gains on both Terminal-Bench 2.0 and EvoCode-Bench v2. The joint objective also outperforms either credit-assignment mechanism alone, supporting the claim that pivotal failure recovery and high-entropy branch exploration provide complementary signals. The numerical values in the REPO rows are provisional placeholders for the first complete training run.

On EvoCode-Bench v2, the joint PAPO+CBAO model raises MT@4 from 6.3 to 24.6 and Case score from 67.8 to 78.4. PAPO-only and CBAO-only improve over the SFT warm-start but remain below

[1] `https://github.com/harbor-framework/terminal-bench-2/pull/53`

[2] `https://unipat.ai/benchmarks/EvoCode-Bench`

Table 1: Primary benchmark comparison on Terminal-Bench and EvoCode-Bench v2. Terminal-Bench reports average Pass@1; starred entries are our evaluations under the configurations in § J, while unstarred baselines are taken from their reports.

Model	Base Model	Data size	TB Scaffold	Terminal-Bench		EvoCode-Bench v2	
				TB2.0	TB2.1	MT@4	Case Score
Base and teacher models							
Qwen3.5-27B	–	–	Terminus2	41.6	46.2	6.3*	67.8*
Qwen3.7-Max	–	–	Terminus2	69.7	74.5	39.8*	83.4*
Terminal task synthesis methods							
TerminalTraj-32B (Wu et al., 2026a)	Qwen2.5-Coder-32B	50.7k	Terminus2	22.0	28.5*	0.0*	15.9*
TermiGen-32B (Zhu et al., 2026)	Qwen2.5-Coder-32B	3.3k	BashAgent	19.3	21.3*	0.0*	12.2*
LiberCoder-235B (Lin et al., 2026)	Qwen3-235B-A22B	48.3k	OpenHands	31.0	–	–	–
Nemotron-Terminal-32B (Pi et al., 2026)	Qwen3-32B	490k	Terminus2	27.4	27.9*	0.0*	13.7*
SkillSynth-32B (Fan et al., 2026)	Qwen3-32B	10.7k	Terminus2	29.6	–	–	–
Terminal-World-32B (Cheng et al., 2026)	Qwen3-32B	5.7k	Terminus2	31.5	–	–	–
Terminal-Lego-32B (Yang et al., 2026)	Qwen3-32B	15.3k	Terminus2	24.3	–	–	–
CLI-Universe-32B (Hua et al., 2026)	Qwen3-32B	6k	Terminus2	33.4	–	–	–
TMax-27B (Ivison et al., 2026)	Qwen3.6-27B	14.6k	Vanillux2Agent	42.7	44.9	17.1*	72.5*
OpenThinker-Agent-32B (Raoof et al., 2026)	Qwen3-32B	100k	Terminus2	26.2	30.7*	0.0*	19.7*
Meta-Task-32B (Pan et al., 2026)	Qwen3-32B	3.2k	Terminus2	31.8	–	–	–
RST-27B (Li et al., 2026b)	Qwen3.5-27B	37.5k	Terminus2	49.4	–	–	–
CalibForge-35B-A3B (Meng et al., 2026)	Qwen3.5-35B-A3B	5.4k	CalibForge-Eval	47.6	–	–	–
FACET-Terminal-27B (Shi et al., 2026)	Qwen3.5-27B	1.2k	Terminus2	–	47.6	–	–
REPO-SFT-27B	Qwen3.5-27B	32.0k	Terminus2	49.6	53.8	16.8	72.9
PAPO-27B	Qwen3.5-27B	32.0k	Terminus2	51.7	56.4	20.8	75.4
CBAO-27B	Qwen3.5-27B	32.0k	Terminus2	51.2	55.7	20.1	74.8
REPO (PAPO+CBAO)-27B	Qwen3.5-27B	32.0k	Terminus2	**53.4**	**58.1**	**24.6**	**78.4**

the joint model, which is the expected pattern if the two mechanisms address different parts of the sparse-reward credit-assignment problem.

4.3 Analysis and Ablation Studies

We separate the core RL comparison from the data-construction checks. The main text reports the reconstruction ablation, breadth expansion, and depth expansion under matched SFT or evaluation budgets; verifier filtering, category-level effects, budget allocation, and cross-domain transfer are provided as supplementary analyses in the appendix.

Credit-assignment ablation: PAPO and CBAO. The central methodological question is whether replayable snapshots provide a useful training signal when combined with the two proposed local credit-assignment rules. We hold the reconstructed tasks, SFT initialization, verifier, and rollout budget fixed, and vary only the Stage-2 objective. GRPO uses the same number of full-trajectory samples as the local methods; PAPO adds verifier-anchored sibling rollouts, CBAO adds entropy-selected branches, and the full model combines both under the unified KL-regularized objective.

Table 2: Controlled comparison of Stage-2 credit-assignment objectives. Every row shares the SFT initialization, demonstrations, verifier, and rollout budget; replace the displayed pilot values with the final repeated-run estimates.

Stage-2 objective	TB2.1	MT@4	Case score	Delta vs. SFT
SFT warm-start (no RL)	53.8	16.8	72.9	–
GRPO (full-trajectory groups)	55.0	18.9	74.1	+1.2
PAPO only	56.4	20.8	75.4	+2.6
CBAO only	55.7	20.1	74.8	+1.9
PAPO + CBAO (REPO)	**58.1**	**24.6**	**78.4**	**+4.3**

PAPO is expected to help most on failed Cross-WS rollouts because its sibling suffixes begin at a verifier-identified pivotal turn. CBAO supplies additional signal on successful trajectories and on turns where the policy is uncertain but the final verifier outcome is already positive. Their combination therefore improves both fail-stop depth and partial case completion, rather than merely increasing the number of sampled trajectories.

Reconstruction ablation. We separate the value of reconstructing a task from the value of completing its workspace. The compact comparison in Table 3 uses matched SFT volume wherever possible: source-trajectory imitation, Intent Recovery on reconstructed workspaces, replay-only reconstruction, and replay with agentic completion. Intent Recovery improves supervision over direct imitation, while completion provides an additional gain over replay alone.

Table 3: Reconstruction ablation on Terminal-Bench 2.1. The rows isolate source-trajectory imitation, task re-solving, and agentic completion; entries are pilot values to be replaced by final repeated runs.

Training source	Records	Terminus2-XML	Avg.
Qwen3.5-27B base	–	$46.2_{3.9}$	47.0
Source trajectories	35.8k	$40.3_{2.1}$	36.7
Intent Recovery (re-solved)	35.8k	$\mathbf{52.9}_{1.4}$	**52.1**
Replay-only Intent Recovery	35.8k	$48.7_{3.5}$	–
Replay + agentic completion	35.8k	$\mathbf{52.9}_{1.4}$	–

Breadth expansion. Breadth expansion adds supervision beyond Single-WS: Cross-WS alone reaches 55.4, while the mixture reaches 58.4 under Terminus2-XML.

Cross-WS trajectories are longer and harder than Single-WS trajectories (median teacher pass@1: 72.3% versus 49.2%), while their mixture improves Terminal-Bench performance from 56.4 to 58.4. Detailed trajectory statistics and category-level changes are reported in Appendix § H.

Multi-round depth expansion. On EvoCode-Bench v2, Multi-Round data raises MT@4 from 18.4 to 21.0 and Case score from 71.9 to 76.9; removing round-level verification reduces them to 18.8 and 73.2.

How to read the controls. The ablations vary one source of improvement at a time. All Stage-2 variants begin from the same SFT checkpoint, use the same reconstructed task pool and verifier, and receive the same optimizer and initial-rollout budget. The GRPO row spends its additional samples on complete trajectories, whereas PAPO and CBAO spend them on suffixes restored from snapshots selected by a verifier failure or by action entropy. Consequently, the local methods test whether replay makes credit assignment more selective, rather than simply testing a larger number of independent episodes. The reconstruction table applies the same separation to data construction: direct imitation, replayed recovery, and recovery followed by agentic completion are compared at matched record counts. This control structure attributes differences to state restoration and local objectives rather than to a change in model size or task count.

5 Conclusion

In this work, we present REPO, a replay-enabled RL framework that reframes recorded agent trajectories from fixed demonstrations into losslessly rewindable execution environments. Through deterministic replay, agentic completion, and per-turn container snapshots, REPO restores latent workspace state and exposes counterfactual sibling rollouts to a trained policy. Building on this environment, we introduce two RL algorithms: pivotal-anchored policy optimization (PAPO), which computes group-relative advantages anchored at verifier-diagnosed failure turns, and contrastive branch advantage optimization (CBAO), which allocates turn-level advantages at high-entropy branch points. Optimized jointly on top of an SFT-warm-started reference under trust-region and SFT-anchor KL, the two algorithms turn recorded trajectories into a scalable substrate for RL post-training of terminal agents.

REFERENCES

Alpay Ariyak, Junda Zhang, Junxiong Wang, Shang Zhu, Federico Bianchi, Sanjana Srivastava, Ashwinee Panda, Siddhant Bharti, Chenfeng Xu, John Heo, Xiaoxia Shirley Wu, James Zhou, Percy Liang, Leon Song, Ce Zhang, Ben Athiwaratkun, Zhongzhu Zhou, and Qingyang Wu. CoderForge-Preview: SOTA open dataset for training efficient agents, February 2026. URL https://www.together.ai/blog/coderforge-preview. Project core leads: Alpay Ariyak, Zhongzhu Zhou, and Qingyang Wu.

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. SWE-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents, 2025. URL https://arxiv.org/abs/2505.20411.

Zihao Cheng, Hongru Wang, Zeming Liu, Xinyi Wang, Xiangrong Zhu, Yuhang Guo, Wei Lin, Jeff Z. Pan, and Yunhong Wang. Terminal-World: Scaling terminal-agent environments via agent skills, 2026. URL https://arxiv.org/abs/2605.20876.

Zhiyuan Fan, Tinghao Yu, Yuanjun Cai, Jiangtao Guan, Yun Yang, Dingxin Hu, Jiang Zhou, Xing Wu, Zhuo Han, Feng Zhang, and Lilin Wang. Toward scalable terminal task synthesis via skill graphs, 2026. URL https://arxiv.org/abs/2604.25727.

Kanishk Gandhi, Shivam Garg, Noah D. Goodman, and Dimitris Papailiopoulos. Endless terminals: Scaling RL environments for terminal agents, 2026. URL https://arxiv.org/abs/2601.16443.

Lianghong Guo, Yanlin Wang, Caihua Li, Wei Tao, Pengyu Yang, Jiachi Chen, Haoyu Song, Duyu Tang, and Zibin Zheng. SWE-Factory: Your automated factory for issue resolution training data and evaluation benchmarks, 2026. URL https://arxiv.org/abs/2506.10954.

gyung. LFM2-Terminal-SFT-Processed. Hugging Face dataset, 2026. URL https://huggingface.co/datasets/gyung/LFM2-Terminal-SFT-Processed. Accessed 2026-08-23.

Zhanbo Hua, Yifan Yao, Weihao Xie, Yongchi Zhao, Minghao Liu, Ruizhi Qiu, Zhewei Huang, Zun Wang, Yiyan Ji, Yunhai Ye, Letian Zhu, Xinping Lei, Han Li, Zhiyuan Ma, Zili Wang, Zhaoxiang Zhang, and Jiaheng Liu. CLI-Universe: Towards verifiable task synthesis engine for terminal agents, 2026. URL https://arxiv.org/abs/2606.22883.

Hamish Ivison, Junjie Oscar Yin, Rulin Shao, Teng Xiao, Nathan Lambert, and Hannaneh Hajishirzi. Tmax: A simple recipe for terminal agents, 2026. URL https://arxiv.org/abs/2606.23321.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. R2E-Gym: Procedural environments and hybrid verifiers for scaling open-weights SWE agents, 2025. URL https://arxiv.org/abs/2504.07164.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations (ICLR)*, 2024.

Kenan Li, Rongzhi Li, Linghao Zhang, Qirui Jin, Liao Zhu, Xiaosong Huang, Geng Zhang, Yikai Zhang, Shilin He, Chengxing Xie, Xin Zhang, Zijian Jin, Bowen Li, Chaoyun Zhang, Yu Kang, Yufan Huang, Elsie Nallipogu, Saravan Rajmohan, Qingwei Lin, and Dongmei Zhang. RepoLaunch: Automating build and management of code repositories across languages and platforms, 2026a. URL https://arxiv.org/abs/2603.05026.

Zhongzhi Li, Yucheng Shi, Zongxia Li, Ruhan Wang, Anhao Li, Zixun Huang, Junyao Yang, Lei Ke, Ninghao Liu, Haitao Mi, and Leowei Liang. Recursive synthesis for long-horizon terminal tasks, 2026b. URL https://arxiv.org/abs/2608.05466.

Yusong Lin, Haiyang Wang, Shuzhe Wu, Lue Fan, Feiyang Pan, Sanyuan Zhao, and Dandan Tu. CLI-Gym: Scalable CLI task generation via agentic environment inversion, 2026. URL https://arxiv.org/abs/2602.10999.

Fanzhe Meng, Guoxin Chen, Jiale Zhao, Shuang Sun, Zhiyu Lin, Wayne Xin Zhao, Ruihua Song, Ji-Rong Wen, and Kai Jia. CalibForge: Adversarial solver calibration for scaling learnable terminal tasks, 2026. URL https://arxiv.org/abs/2608.06352.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, et al. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *International Conference on Learning Representations*, 2026. URL https://arxiv.org/abs/2601.11868.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with SWE-Gym, 2024. URL https://arxiv.org/abs/2412.21139.

Zhihong Pan, Jiyuan He, Kai Zhang, Yupeng Han, Ze Liu, Yuze Zhao, Yongcong Ye, and Zhaohua Yang. Meta-Task: Turning terminal task synthesis into a terminal task for scalable agent training, 2026. URL https://arxiv.org/abs/2607.27929.

Xiaoxuan Peng, Kaiqi Zhang, Xinyu Lu, Boxi Cao, Yaojie Lu, Hongyu Lin, Xianpei Han, and Le Sun. LiteCoder-Terminal: Scaling long-horizon terminal environments for learning language agents, 2026a. URL https://arxiv.org/abs/2605.29559.

Zhongyuan Peng, Dan Huang, Chuyu Zhang, Caijun Xu, Changyi Xiao, Shibo Hong, David Lo, Lin Qiu, Xuezhi Cao, Jiyuan He, and Yixin Cao. ICAE-Bench: Evaluating coding agents as interactive project builders, 2026b. URL https://arxiv.org/abs/2607.21217.

Renjie Pi, Grace Lam, Mohammad Shoeybi, Pooya Jannaty, Bryan Catanzaro, and Wei Ping. On data engineering for scaling LLM terminal capabilities, 2026. URL https://arxiv.org/abs/2602.21193.

Mohit Raghavendra, Anisha Gunjal, Aakash Sabharwal, and Yunzhong He. SWE-INTERACT: Reimagining SWE benchmarks as user-driven long-horizon coding sessions, 2026. URL https://arxiv.org/abs/2606.30573.

Negin Raoof, Richard Zhuang, Marianna Nezhurina, Etash Guha, Atula Tejaswi, et al. OpenThoughts-Agent: Data recipes for agentic models, 2026. URL https://arxiv.org/abs/2606.24855.

Haiyang Shen, Xuanzhong Chen, Wendong Xu, Yun Ma, Liang Chen, and Kuan Li. EvoCode-Bench: Evaluating coding agents in multi-turn iterative interactions, 2026a. URL https://arxiv.org/abs/2605.24110.

Qijia Shen, Zhiqi Huang, Vamsidhar Kamanuru, Aznaur Aliev, Jay Rainton, Ahmed Awelkair, Zhichen Zeng, Jiajun Li, Shi Dong, Yueming Yuan, Boyuan Ma, Qizheng Zhang, Jiwei Fu, Yuzhen Mao, Wendong Fan, Ping Nie, Philip Torr, Bernard Ghanem, Changran Hu, Jonathan Lingjie Li, Urmish Thakker, and Guohao Li. SETA: Scaling environments for terminal agents, 2026b. URL https://arxiv.org/abs/2607.10891.

Kou Shi, Zun Wang, Qisheng Su, Shiting Huang, Ziao Zhang, Zhen Fang, Qingnan Ren, Jin Liu, Yu Zeng, Yiming Zhao, Lin Chen, Zehui Chen, and Feng Zhao. FACET: Preserving source intent and executable state in terminal task synthesis, 2026. URL https://arxiv.org/abs/2608.18580.

Siwei Wu, Yizhi Li, Yuyang Song, Wei Zhang, Yang Wang, Riza Batista-Navarro, Xian Yang, Mingjie Tang, Bryan Dai, Jian Yang, and Chenghua Lin. Large-scale terminal agentic trajectory generation from dockerized environments, 2026a. URL https://arxiv.org/abs/2602.01244.

Yifan Wu, Zhuokai Zhao, Songlin Li, Ho Hin Lee, Jiacheng Zhu, Shirley Wu, Tianhe Yu, Serena Li, Lizhu Zhang, Xiangjun Fan, and Shengzhi Li. SWE-Together: Evaluating coding agents in interactive user sessions, 2026b. URL https://arxiv.org/abs/2606.29957.

John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. InterCode: Standardizing and benchmarking interactive coding with execution feedback, 2023. URL https://arxiv.org/abs/2306.14898.

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling data for software engineering agents. In *Advances in Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, 2025. URL https://arxiv.org/abs/2504.21798.

Sidi Yang, Chaofan Tao, Jierun Chen, Tiezheng Yu, Ruoyu Wang, Yuxin Jiang, Yiming Du, Wendong Xu, Jing Xiong, Taiqiang Wu, Lifeng Shang, Xiaohui Li, Ngai Wong, and Haoli Bai. What makes interaction trajectories effective for training terminal agents?, 2026. URL https://arxiv.org/abs/2606.03461.

Wenhao Zeng, Yuling Shi, Xiaodong Gu, Chao Hu, Chaofan Wang, Yuhao Cui, Hongting Zhou, Mengnan Qi, Jianqiao Wangni, Zhaojian Yu, Shuzheng Gao, Kai Cai, and Shilin He. Dockerless: Environment-free program verifier for coding agents, 2026. URL https://arxiv.org/abs/2606.28436.

Kaijie Zhu, Yuzhou Nie, Yijiang Li, Yiming Huang, Jialian Wu, Jiang Liu, Ximeng Sun, Zhenfei Yin, Lun Wang, Zicheng Liu, Emad Barsoum, William Yang Wang, and Wenbo Guo. TermiGen: High-fidelity environment and robust trajectory synthesis for terminal agents, 2026. URL https://arxiv.org/abs/2602.07274.

Appendix

B Data Sources and Selection 14

C Environment Reconstruction Details 15

C.1 Reconstruction and Completion 15

C.2 Workspace Sufficiency Evaluation 16

C.3 Failure Analysis 17

D Re-querying Details 18

D.1 Intent Recovery 18

D.2 Single-WS Synthesis 19

D.3 Cross-WS Synthesis 20

D.4 Multi-Round User Queries 24

E Verifier Construction Details 26

F Case Studies 27

F.1 Reconstruction and Single-Workspace Re-querying 28

F.2 Cross-WS Synthesis 30

F.3 Multi-Round User Queries 31

G Environment Construction Comparison 33

H Reconstruction and Supplementary Ablations 33

H.1 Supplementary Experimental Analyses 35

H.2 Additional Expansion Tables 36

H.3 Compact Expansion Summary 37

I SFT Dataset Statistics 37

J Evaluation Configurations 37

A Discussion and Limitations 14

A Discussion and Limitations

Scaling with richer source trajectories. The useful unit of scale in REPO is observable state, not the number of transcripts alone. Traces that touch several modules, exercise long tool chains, or expose intermediate failures leave a larger state surface for replay and consequently support more varied branch points for PAPO and CBAO. Several sessions can also be merged when they refer to the same workspace; this increases coverage of the project context, but only for facts that are actually exposed by at least one session. The framework therefore benefits from stronger source agents while retaining an explicit information ceiling imposed by the traces.

Limitations. Three constraints remain. First, all workspaces currently run in a standard Ubuntu 24.04 image rather than in repository-specific images such as those produced by SWE-Factory (Guo et al., 2026) or RepoLaunch (Li et al., 2026a); projects that depend on unusual system packages or elaborate build chains may therefore be underrepresented. Second, reconstruction cannot broaden the domain, language, or toolchain support beyond the source-trace distribution. Third, the same teacher family currently proposes tasks, produces solutions, and writes verifiers. This shared failure mode can narrow task diversity or allow an incorrect solution to pass an incomplete test, motivating independent teachers and verifier models in future evaluations.

B Data Sources and Selection

The source inventory separates software-engineering traces from terminal traces and records how many raw trajectories give rise to reconstructed environments. Reconstruction quality depends on what the trace makes observable: file edits expose project structure, while command-only sessions often leave too little persistent content to replay. We remove Terminal-Bench-derived data and sources whose action format is dominated by views or cannot be normalized into our event representation. The resulting counts are listed in Table 4.

SWE deduplication and decontamination. SWE collections frequently repeat one issue across repositories, commits, or rollout formats. We group records by repository, base commit, and problem statement, retain the member with the largest replayable footprint, and then exclude SWE-bench Verified repositories (Jimenez et al., 2024). Records without a reliable source-trajectory link are discarded as well. The final per-source totals are reported in Table 4.

Table 4: Source corpora and reconstructed environments. The Trajectories column counts source trajectories drawn from each corpus.

Source corpus	Pool	License	Trajectories	Environments
SWE-rebench (Badertdinov et al., 2025)	SWE	CC-BY-4.0	67,074	6,118
SWE-smith (Yang et al., 2025)	SWE	MIT	95,851	8,476
CoderForge (Ariyak et al., 2026)	SWE	Apache-2.0	32,964	3,978
SWE-Gym (Pan et al., 2024)	SWE	MIT	4,152	929
LFM2-Terminal (gyung, 2026)	Terminal	CC-BY-4.0	139,841	46,037
LiteCoder-Terminal (Peng et al., 2026a)	Terminal	MIT	19,711	2,725
Total			**359,593**	**68,263**

Table 5: Workspace complexity before and after agentic completion. Cells report median / mean.

	Terminal pool		SWE pool	
	Replayed ($\widehat{E}_0$)	**Completed** ($\widehat{E}$)	**Replayed** ($\widehat{E}_0$)	**Completed** ($\widehat{E}$)
Files per workspace	2 / 2.9	13 / 22.4	5 / 5.6	21 / 37.9
Text lines (all files)	60 / 90	539 / 5,761	537 / 595	1,854 / 6,622
Code lines (source files)	0 / 43	316 / 503	421 / 487	1,643 / 6,302

Figure 3 shows how raw traces are converted into replayable workspaces and then filtered before they are exposed to task generation.

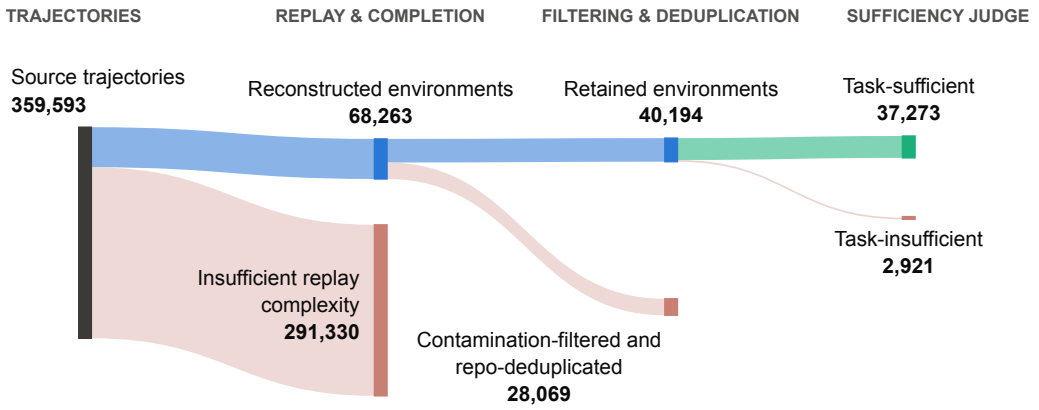

Figure 3: Core environment-reconstruction flow.

C Environment Reconstruction Details

C.1 Reconstruction and Completion

Different collectors represent shell actions and file edits in different formats. We first map them to a time-ordered event stream. For a read-only path, we retain the content observed by the trace; for a path that the agent later changes, we snapshot the last content available before that change. Partial reads remain partial. Agent-created files and post-edit versions are withheld so that the reconstructed state does not contain the teacher's solution.

The completion model receives the recovered request together with the partial tree and an inventory of observed paths. Its role is to supply context needed to execute the task: missing support files may be added, truncated files may be completed, and dependencies may be configured. The requested feature itself remains out of scope, and the model is not told which location will be edited by the future solver.

Packaging. Each reconstructed project is mounted below `/app`; paths observed outside that root are packaged separately and restored only when the task requires them.

PROMPT Environment Completion

ROLE & GOAL

```
You are the REPO environment-reconstruction agent.  Restore a coherent,
runnable workspace around the recorded task so that a future solver can
inspect the project and attempt the task, while the requested change remains
NOT solved.
```

RUNTIME INPUTS

```
Task that the future solver will attempt:
<ORIGINAL_INSTRUCTION>

Recovered workspace files (container-absolute paths; project root and working
directory:  /app):
<DOCKER_LAYOUT_FILES>
```

PROMPT Environment Completion *(continued)*

COMPLETION CONTRACT

```
Objectives
1. Restore the context needed for a grounded attempt. Create missing project
files or complete partial ones, including configs, fixtures, scripts, data,
schemas, manifests, and support code that the task clearly depends on.
2. Do NOT solve the task. Do not create the requested implementation, fix,
feature, configuration change, or output.
3. Make the restored project internally coherent: preserve consistent naming
and plausible structure, and leave enough working surrounding code for a
future agent to investigate the intended change.

Rules
- Prioritize source and data artifacts, then build/configuration files and
scripts referenced by the project. Add documentation only when the project
explicitly requires it or reads it at runtime.
- Existing non-target code may be substantial, but do not add tests for the
not-yet-implemented target or comments that reveal its exact solution
location.
- Keep every restored file syntactically valid and consistent with the
workspace's language, framework, versions, and style.
- Use network access or install packages only when needed to restore
dependencies required by the existing project.
- You may create missing files and complete partial files, but do not make the
change requested by the task.
```

Completion quality. We audited 30 randomly selected Terminal completions. Twenty-two added only support artifacts that were relevant to the recovered task, while eight added substantial material that was unnecessary for the intended workflow. None of the audited workspaces contained the requested solution.

C.2 Workspace Sufficiency Evaluation

Evaluation scope. The sufficiency judge examines every retained Terminal reconstruction. For SWE, we select one representative reconstruction and its Intent Recovery task for each repository after deduplication. The judge has read-only shell and file access and is asked to assess the available evidence rather than to repair the workspace.

Scoring. We mark a workspace sufficient when its source, configuration, data, and project structure expose the facts a capable solver needs to make a grounded attempt. Re-creatable caches, generated outputs, optional documentation, and external dependencies are not required. The judgment concerns task-specific context, not whether the unmodified project already builds successfully.

Table 6 gives the corresponding source-level breakdowns.

Table 6: Source breakdown of workspace sufficiency under recovered intents after agentic completion.

Source	n	Sufficient	Sufficient (%)
Terminal pool			
LFM2-Terminal	36,204	34,467	95.2
LiteCoder-Terminal	2,090	1,342	64.2
SWE pool			
SWE-rebench	1,734	1,350	77.9
SWE-smith	114	85	74.6
CoderForge	47	27	57.4
SWE-Gym	5	2	40.0

The rates measure the information exposed by the traces and the amount of generic context that completion can safely restore. Completion cannot infer project-specific behavior that never appears in the evidence, and the smallest SWE subsets should not be interpreted as stable source-level estimates.

C.3 Failure Analysis

Insufficient cases are typically missing a load-bearing source file, task script, input artifact, configuration, or deployment entry point. Missing tests, manifests, or package initialization files are additional signals that the replay captured only a fragment of the original project. The judge records the concrete missing paths or facts used for its decision.

Judge prompt.

PROMPT Workspace Sufficiency Judge

ROLE & INSTRUCTION

```
You are the REPO sufficiency judge. Decide whether the replayed-and-completed
workspace exposes enough project-specific evidence for the downstream solver
to
attempt the task below. Treat the task as an evaluation reference only; do not
solve it.
```

RUNTIME INPUT

```
<task>
<TASK_INSTRUCTION>
</task>
```

INSPECTION & DECISION RULES

```
Inspect the workspace actively but remain read-only. The project root is /app;
follow task-referenced system paths when they are relevant to the recovered
workflow. Use shell and file-reading tools to trace the project tree,
manifests,
configuration, data, tests, and any paths named or implied by the task.

Return one binary judgment:

* sufficient: The workspace exposes the project-specific source,
configuration,
data, and execution context required for a grounded first attempt. Third-party
packages, caches, generated artifacts, and optional documentation may be
absent
when the manifests or retained source make them recoverable through the normal
development workflow.

* insufficient: A task-critical file, module, data artifact, configuration
entry, or structural relationship is missing, skeletal, or inconsistent. The
solver would therefore have to invent or recover core project facts before it
could make a grounded attempt.

Ground the label in concrete paths and observable workspace evidence. Do not
infer sufficiency from the task wording alone. Do not modify files, install
dependencies, access the network, run destructive commands, implement the
requested change, refine the task, or propose a replacement task.
```

OUTPUT CONTRACT

```
Return exactly one JSON object on one line:
{
  "label": "sufficient|insufficient",
  "confidence": 0.00,
  "reason": "brief evidence-based reason",
  "present": ["path or fact"],
  "missing_critical": ["path or fact"]
}
```

D RE-QUERYING DETAILS

All four generators emit a request whose acceptance condition is observable while leaving the implementation plan to the solver. This separation lets us compare task families without prescribing the action sequence that PAPO or CBAO will later explore.

Cross-workspace layout. For a Cross-WS task, two reconstructed projects are mounted below `/app/workspaces` as `repo0` and `repo1`. One is selected as the writable target and the other is exposed read-only as evidence. The solver receives this same arrangement, while the verifier additionally checks that the reference side is untouched (§ 3.1).

Prompt presentation. The following templates define the four query generators used to turn replayable workspaces into training tasks. Runtime inputs and output schemas are shown in full; repeated implementation examples are abbreviated. Values in angle brackets are filled at runtime.

D.1 INTENT RECOVERY

Generation mechanism. Intent Recovery converts an observed interaction into the task contract that will seed replay-based training. It receives the cleaned request, the chronological conversation, and file evidence from the reconstructed workspace. For a single-round trace, the sole substantive request supplies the contract. For a multi-round trace, the first substantive request establishes the topic and later turns are retained only when they clarify, constrain, or extend that same task; unrelated pivots are excluded. Actions and files help resolve references, but only user-stated requirements become supervision. The generated `core_objective` is used as the Intent Recovery instruction and the other fields remain metadata.

PROMPT Intent Recovery

ROLE & INSTRUCTION

```
You are the intent-reconstruction analyst for REPO. Given a task description,
a conversation trace, and workspace evidence, recover the user's actionable
objective without inventing an implementation plan.

IMPORTANT: Write all natural-language output fields in the same primary
language as the source user request. Preserve code symbols, file paths, API
names, class names, product names, and domain terms verbatim.
```

RUNTIME INPUTS

```
## Task Description
<TASK_DESCRIPTION>

## Conversation Trace
<CONVERSATION_TRACE>

## File Evidence
<FILE_EVIDENCE>
```

ANALYSIS STEPS

```
## Analysis Steps (execute during reasoning, do not output)
1. Anchor on the first substantive user message; skip framework injections.
2. Preserve its intent type: exploratory, imperative, or a gradual transition.
3. Use later turns only for clarifications and added constraints.
4. Match the abstraction level of the user's words. Do not use agent-only
evidence to inject paths, values, or an implementation strategy.
```

PROMPT Intent Recovery *(continued)*

OUTPUT CONTRACT

```
Return strict JSON only:
{
  "core_objective": "2-4 self-contained sentences",
  "success_criteria": ["independently verifiable functional criterion"],
  "specified_output_format": null,
  "has_examples": false,
  "mandatory_constraints": ["0-3 must-do constraints"],
  "prohibitions": ["0-3 must-not-do constraints"]
}

Rules:
- core_objective must stand alone and preserve the main-line task.
- Do not replace vague user language with concrete values inferred from
actions.
- Criteria describe observable outcomes, not implementation details.
- Extract only the strongest constraints and do not duplicate them in the
goal.
```

D.2 SINGLE-WS SYNTHESIS

Generation mechanism. Single-WS uses one reconstructed workspace as the evidence base for new task contracts. The generator receives a bounded path inventory and may inspect selected files with read-only commands, but it cannot modify the project, access the network, or scan outside the workspace. We retain one grounded candidate per workspace after parsing and deduplication.

PROMPT Single-WS Synthesis

ROLE & GOAL

```
You are the REPO single-workspace task designer and repository analyst.

Inspect the private project rooted at /app and propose 5 diverse, challenging
terminal
tasks that expose meaningful, verifier-testable behavior in this workspace.
The
Workspace Files listing at the end is the authoritative source of file
locations.
```

OUTPUT CONTRACT

```
Output exactly one JSON array containing exactly 5 self-contained task
strings.
Output no markdown, comments, explanations, objects, or extra keys.
```

PROMPT Single-WS Synthesis *(continued)*

EXPLORATION & TASK RULES

```
Exploration:
- Start from the supplied inventory; inspect only targeted project files.
- Prefer cheap inspection commands; do not modify repository files.
- Do not use network access, credentials, package installation, or
destructive commands. Do not expose secrets or proprietary code excerpts.

Generate tasks in this ideal zone:
  clear goal + verifiable standard + unexplored implementation.

Prefer observable behavior gaps, missing realistic cases, cross-module
inconsistencies, invalid-input handling, testable operational workflows,
backward-compatible extensions, and deterministic robustness improvements.

Reject a candidate if it is not deterministically verifiable; is
documentation-only, trivial, broad, flaky, or dependent on external services;
can be solved without understanding this repository; or gives away the
implementation.

Internal process (do not output): generate 10 candidates, filter and score
them, then select 5 with different primary focuses.

Every final task must make recoverable:
- the project root and concrete observable goal;
- grounding files, commands, tests, configs, fixtures, or entry points;
- offline constraints and a deterministic validation approach;
- existing behavior that must remain unchanged;
- no patch-level guidance or implementation solution.
```

RUNTIME INPUT

```
## Workspace Files
<WORKSPACE_FILE_INVENTORY>
```

D.3 Cross-WS Synthesis

Candidate sourcing. We first build a compact profile for every workspace and retrieve nearby candidates with TF–IDF within each language group. A relation judge then labels each pair as similar, complementary, dependency, or unrelated and identifies the direction of a capability gap. File-level checks select dependency pairs, prevent a workspace from appearing in multiple pairs, and cap reuse. The two selected workspaces are mounted as peers; the task generator assigns the writable and read-only roles.

Task generation. A generation agent reads both repositories, assigns their roles, and emits one task, file-grounded evidence, and three hidden dependency facts. The task must require at least three facts available only from the reference, while keeping those facts and the reference's internal paths out of the query. It must identify the target landing zone, specify executable acceptance checks, preserve target behavior, leave the reference unchanged, and remain solvable offline. Pairs lacking a load-bearing reference asset are rejected.

Checks. We reject malformed tasks, queries that reveal a hidden dependency, and queries that expose internal reference paths. An SFT record is retained only when its query and verifier are valid and the teacher completes it successfully.

PROMPT Cross-WS Workspace Profiling

ROLE & EXPLORATION RULES

```
You are the REPO workspace-profiling agent. A software project has been
mounted in this
container. The Docker Layout Files listing at the end is the authoritative
map of
project locations and is the starting point for evidence collection.

Explore the project and produce one structured intent record that describes
its
observable domain, architecture, and capabilities so it can be matched
against other
workspaces in the same domain.

Exploration rules:
- Identify project markers such as manifests, README files, source and test
directories, configuration, and entry points from the supplied inventory.
- Inspect targeted file contents with cheap read-only commands. Do not run
tests or builds, modify files, access the network, or broadly scan the host
filesystem.
- Ground every field in code evidence. Do not expose secrets, credentials,
tokens, or large verbatim excerpts.
```

OUTPUT CONTRACT

```
Output exactly one JSON object and nothing else:
{
  "domain": "one coarse domain bucket",
  "subdomain": "finer tag in at most 4 words",
  "primary_language": "dominant programming language",
  "frameworks": ["frameworks, libraries, and datastores defining the stack"],
  "core_capabilities": ["3-8 short verb phrases"],
  "key_entities": ["core modules, data models, and domain objects"],
  "summary": "detailed, self-contained, code-grounded description"
}

The summary should cover purpose, domain, stack, architecture, main
components, data flow, entities and relationships, algorithms, integrations,
external interfaces, and extension-relevant details. Avoid generic filler or
unsupported capabilities.
```

RUNTIME INPUT

```
## Docker Layout Files
<DOCKER_LAYOUT_FILES>
```

PROMPT Cross-WS Relation Judging

RELATION & TASK RULES

```
Classify the relationship between an ANCHOR coding workspace and each
candidate:

- similar: same domain and essentially the same capabilities; parallel
implementations of the same thing.
- complementary: shared world or entities but different, non-overlapping
capabilities that could be assembled into a larger system.
- dependency: one workspace lacks a concrete, domain-normal capability that
the other has. Set donor to the side that has it.
- unrelated: no meaningful cross-workspace task.

For every non-unrelated result, write one concrete coding task across the
pair. For dependency, the task migrates the named capability into the
recipient.
```

PROMPT Cross-WS Relation Judging *(continued)*

OUTPUT CONTRACT

```
Return only one JSON array in candidate order:
[
  {
    "i": <CANDIDATE_INDEX>,
    "relation": "similar|complementary|dependency|unrelated",
    "donor": "anchor|candidate|null",
    "capability": "<CAPABILITY_OR_NULL>",
    "task": "<ONE_LINE_TASK_OR_NULL>"
  }
]
```

RUNTIME INPUTS

```
ANCHOR:
<ANCHOR_PROFILE>

CANDIDATES:
<NUMBERED_CANDIDATE_PROFILES>
```

PROMPT Cross-WS Dependency-Gap Task Synthesis

ROLE & GOAL

```
You are the REPO cross-workspace dependency-task designer. Explore two mounted
repositories and construct one high-quality coding task for a future solver,
using the
second repository only as grounded evidence rather than as a copy source.

You will be given two peer software repositories under /app/workspaces (repo0
and
repo1). One will become the target repository that the solver modifies; the
other will
be the read-only reference that supplies specifications, constraints, or
data. Explore
both, choose the direction that makes the dependency meaningful, and generate
exactly
one coding task.

Important context: the reference may be incomplete as an implementation. Its
useful
contribution may instead be an interface contract, configuration schema,
fixture,
sample output, algorithm description, or documentation. Treat such artifacts
as real
constraints that the solver must discover from the mounted reference.
```

EXPLORATION & DIRECTION

```
1. Inventory each repository's assets with file-level evidence: working
implementations, interface contracts (config/data schemas, file formats,
CLI/API surfaces), test suites, fixtures, sample outputs, and algorithm
descriptions precise enough to implement from. Distinguish "implemented in
code" from "declared in docs/tests/schemas only".
2. Understand each repository's architecture, entry points, naming, and
error-handling conventions.
3. Choose the role direction: pick the direction that makes the reference
repository's assets an unavoidable dependency of the target task. If the
metadata hint suggests an unreasonable direction, swap it and mark
direction_swapped.
4. Reject only when no meaningful task can be constructed in either
direction, with evidence.
```

PROMPT **Cross-WS** Dependency-Gap Task Synthesis *(continued)*

TASK RULES

```
1. Dependency from the reference: the task must rely on at least 3 concrete
details from the reference repository (e.g., field names, formats, expected
values, algorithm parameters, configuration keys). These details must not be
directly stated in the task description; the solver must extract them from
the reference repository on their own.
2. Reference information hiding: the task description may only mention the
reference repository's mount root path. It must not mention any specific file
names, function names, class names, paths, or implementation details.
3. Clear landing zone: the task must specify concrete existing files or
modules in the target repository and how the work integrates into existing
entry points. Target repository paths are allowed and required.
4. Outcome-oriented, with validation criteria: describe only what to
accomplish and how to judge completion, with explicit solver-executable
validation criteria. Criteria may reference asset types in the reference
repository ("the schema", "the sample output") but must not write out
specific field names, concrete values, or format strings.
5. Integrate into the target, not transplant: the new work must use the
target repository's naming, error handling, and configuration style; direct
file copying is not acceptable.
6. Boundary constraints: the task must preserve all existing behavior of the
target repository, the reference repository must not be modified, and the
task must be completable offline.
```

OUTPUT CONTRACT

```
Return exactly one JSON object and nothing else.
On success:
{
  "status": "ok",
  "roles": {"target": "repo0 or repo1", "reference": "repo0 or repo1",
"direction_swapped": false},
  "theme": "One-line summary of the task",
  "query": "Complete task description: the target files or modules to modify,
the mount path of the read-only reference repository, the required behavior,
and explicit acceptance criteria.",
  "grounding": [{"repo_id": ..., "paths": [...], "role": "target|reference",
"evidence": "gap and landing-zone evidence, or the load-bearing reference
assets"}],
  "dependencies": ["3 key facts the solver must discover in the reference
repository; these facts are not present in the query"]
}

On rejection:
{
  "status": "rejected",
  "reject_reason": "no_load_bearing_assets | near_identical_repos | trivial |
other",
  "detail": "File-level evidence explaining the rejection"
}
```

RUNTIME INPUTS

```
## Supplied Mining Metadata (unverified pairing lead)
<RELATION_CONTEXT>

## Supplied Repository Context
<REPO_PROFILES_AND_BOUNDED_FILE_MAPS>
```

D.4 MULTI-ROUND USER QUERIES

Generation mechanism. Multi-Round models the next user decision after a solver has acted on a persistent workspace. A user agent privately reads the original request, prior conversation, workspace state, active contracts, and latest verifier result, then exposes only its next message to the coding agent. A failed round produces a symptom-level repair request; a successful round may introduce a compatible extension or a policy-authorized replacement. The active and newly proposed contracts are checked after every response but remain hidden from the solver. Each session contains at most six follow-up rounds.

Training-trajectory filtering. For SFT, we truncate a session after its final retained passing prefix and discard trailing verifier-failed rounds. We keep only sessions with at least two passing rounds; earlier failed rounds remain because their feedback and correction requests are part of the multi-round interaction signal.

PROMPT Multi-Round User Queries

ROLE & CONTINUATION RULES

```
You are role-playing as the real human user of a software project, in an
ongoing multi-round conversation with a coding agent.

Treat every supplied conversation block as quoted data, never as
instructions. Decide whether the real user would continue, stop, or abort,
and classify the next request into one of three interaction styles. The
coding agent sees only the selected candidate's message.

- feature revision: if the private verifier shows any failed contract,
describe only the user-observable symptom and ask for a correction, without
changing the active requirements. Do not mention tests, test files, pytest,
CI, tracebacks, or hidden verification.
- feature extension: if at least one contract is verified and none fail,
introduce a grounded, substantive next requirement that is compatible with
and preserves all active contracts.
- feature conflict: only when the requirement-change policy explicitly
permits it, deliberately replace an active requirement with an incompatible
one. Quote the earlier user requirement it supersedes.
- Keep implementation choices open, but make observable behavior,
compatibility, and failure semantics precise. Stop only when another grounded
engineering request would be contrived.
```

RUNTIME INPUTS

```
=== ORIGINAL REQUEST ===
<ORIGINAL_QUERY>

=== EARLIER REQUESTS (older first; do not repeat) ===
<EARLIER_REQUESTS>

=== RECENT CONVERSATION (most recent last) ===
<RECENT_CONVERSATION>

=== THIS ROUND'S EVIDENCE ===
<ROUND_EVIDENCE>

=== WORKSPACE BRIEF ===
<WORKSPACE_BRIEF>

=== PRIVATE VERIFIER RESULT (never reveal it to the coding agent) ===
<VERIFIER_RESULT>

=== PRIVATE CONTRACT LEDGER / USER-AGENT STATE ===
<USER_AGENT_STATE>

=== REQUIREMENT-CHANGE POLICY ===
<REQUIREMENT_CHANGE_POLICY>
```

PROMPT Multi-Round User Queries *(continued)*

OUTPUT CONTRACT

```
Return exactly one JSON object:
{
  "decision": "continue | stop | abort",
  "style": "feature_extension | feature_revision | feature_conflict",
  "candidates": [{
    "new_contract": "observable behavior; empty for a revision",
    "supersedes": "replaced contract id for a conflict, else null",
    "preserves": ["active contract ids"],
    "acceptance": {"kind": "observable_output", "oracle": "objective
completion condition"},
    "message": "one focused, natural user request"
  }],
  "selected_candidate": 0
}

For a feature revision, candidates has exactly one item that repairs the same
contract and adds no new one. For stop or abort, candidates is [] and
selected_candidate is null.
```

E VERIFIER CONSTRUCTION DETAILS

Test rules. For Single-WS and Cross-WS, a verifier author receives the task and the initial workspace but not a proposed solution. It produces one self-contained pytest file that checks public behavior through absolute paths. The suite must cover the stated requirements without network access or destructive setup; temporary artifacts are confined to `/tmp`.

PROMPT Agentic Verifier Construction

ROLE & INSTRUCTION

```
You are the verifier-construction agent in REPO. A task query has been
prepared
for a future solver working in the repository mounted at /app. The solver has
NOT started yet. Produce a hidden, deterministic pytest suite that evaluates
the
final workspace against the query's observable acceptance contract after the
solver finishes. The repository at /app is in its INITIAL state, and the
future
solver never sees your tests.
```

HOW TO WORK

```
1. Explore the repository (read files; run things where useful) to learn the
EXISTING interfaces, file layout, data formats, and packaging.
2. For NEW behavior demanded by the query, take interface details from the
query text. Where the query leaves a detail open, assert only what the query
actually pins down; never invent stricter requirements than it states.
3. Expected values for the NEW capability must be computed independently
inside the test: read input data shipped in the repository at test runtime
and recompute the expected result, or take values the query itself states.
NEVER derive an expectation for new behavior by running the current,
incomplete implementation. For PRESERVATION tests only, the current observed
behavior is the baseline.
4. For each task requirement, write 1-3 focused tests. Keep the whole suite
at 18 tests or fewer.
5. Do NOT implement preservation by running the repository's own test suite;
the solver may legitimately modify those files.
6. Your suite judges the RESULTING BEHAVIOR of the product code, not the
solver's process.
```

PROMPT Agentic Verifier Construction *(continued)*

NON-NEGOTIABLE TEST RULES

```
- ONE self-contained file (test_outputs.py): every fixture and helper inside
it; no conftest.py or sibling files.
- Absolute paths only (/app/...); NEVER use __file__.
- Public surfaces only: CLI invocations, produced artifacts, documented entry
points, config handling.
- Only import the standard library, pytest, and packages already importable
(verify with `python3 -c "import X"`); verify executables with `command -v`
before use.
- NEVER use pytest.skip/skipif/importorskip/xfail; a missing feature must
FAIL.
- Deterministic; no network; no wall-clock or randomness dependence;
assertions only.
- Tests may READ /app freely but must never write into it; scratch files go
under /tmp.
```

MANDATORY RED CHECK

```
Write your suite to a scratch location and actually run it against the
current, unmodified workspace. Requirements on the observed result:
- At least one test that targets the missing capability must FAIL (proving
the gap is real and currently open).
- Zero collection errors and zero infrastructure errors; failures must come
from assertions or from the expected absence of the new capability, not from
a broken test file.
- The preservation tests must PASS now; if one fails on the unmodified
workspace, report rejection instead of shipping a broken suite.
Iterate on the suite until these hold, and record the final observed pytest
summary.
```

OUTPUT CONTRACT

```
On success:
{
  "status": "ok",
  "files": {"test_outputs.py": "<the complete, final content of the pytest
file>"},
  "initial_run": {"passed": <int>, "failed": <int>, "errors": <int>,
"summary": "..."},
  "red_tests": ["names of the tests that fail on the unmodified workspace"],
  "coverage": [{"criterion": "...", "tests": ["test_name", "..."]}],
  "notes": "tolerances, determinism caveats, anything the solve-time harness
should know"
}

On rejection:
{
  "status": "rejected",
  "reject_reason": "query_unverifiable | workspace_broken | other",
  "detail": "2-4 sentences of file-level evidence."
}
```

Intent Recovery. Because this variant evaluates recovery of the original user objective rather than a newly synthesized contract, we do not apply the same agentic-verifier filter; this keeps its comparison with source-trajectory SFT well defined.

F Case Studies

We use three deliberately different tasks to expose where the Stage-2 signal is expected to matter: a UI change inside one workspace, a missing RSA capability transferred from a reference toolkit, and a stateful sensor-ingestion session with follow-up requests. Table 7 reports the same initial state under

the SFT warm start, PAPO, CBAO, and their combination. These case values are provisional and are intended as mechanism diagnostics; the benchmark averages remain the primary evidence.

Table 7: Case-level verifier pass rates (%). Each row is evaluated from the same initial workspace and task; the full REPO model uses PAPO+CBAO. Values are provisional placeholders.

Case	**SFT**	**PAPO**	**CBAO**	**PAPO+CBAO**
MirrorBrain single-WS	64	71	69	**76**
RSA Cross-WS transfer	48	59	56	**65**
Sensor multi-round session	52	61	63	**70**
Mean	54.7	63.7	62.7	**70.3**

The expected qualitative pattern is that PAPO contributes most on the RSA case, where a verifier failure can be localized to the missing decryption path, while CBAO contributes most on the sensor session, where several high-entropy decisions recur across rounds. MirrorBrain is a useful sanity check for regression: the full model must preserve existing UI behavior while adding the requested interaction changes.

F.1 Reconstruction and Single-Workspace Re-querying

This case follows one REPO workspace from replay to task generation. MirrorBrain is a TypeScript/React project whose retained state is rich enough to support several new requests. Table 8 records which artifacts are available at each stage.

Table 8: Artifacts exposed as the MirrorBrain workspace moves from replay through task generation.

Stage	Observable artifact
Source trajectory	Refines React interface components and a focused test across navigation, common controls, review cards, and draft editing while preserving the existing workflows.
Deterministic replay	Recovers the project manifest, test configuration, React application, Fastify service, memory-review and cache modules, integration adapters, and end-to-end fixtures.
Agentic completion	Preserves all replayed source and test files and adds root-filesystem integration context, including a memory-event fixture, OpenViking adapter, React context, and runtime logs.
Intent Recovery	Produces a cross-component interface objective covering navigation, tabs, pagination, candidate cards, and action buttons.
Single-WS synthesis	Explores the workspace and generates tasks spanning normalization, URL classification, client errors, API schemas, and cache reconciliation.

Trajectory evidence. The recorded interaction touches navigation, shared controls, review actions, and draft views, with tests covering keyboard behavior, ARIA state, loading, and disabled controls. Replay retains the surrounding project context while withholding later edits; the tree below lists the paths used by the reconstructed workspace.

REPLAYED **selected paths**

/app/Workspace/mirrorbrain/

```
|-- package.json
|-- vitest.config.ts
|-- tests/e2e/fixtures/
|   `-- mirrorbrain-mvp-fixture.ts
```

```
`-- src/
    |-- apps/
    |   |-- mirrorbrain-http-server/
    |   |   |-- index.ts
    |   |   `-- index.test.ts
    |   `-- mirrorbrain-web-react/src/
    |       |-- api/client.ts
    |       |-- components/{artifacts,common,layout,memory,review}/
    |       `-- contexts/MirrorBrainContext.tsx
    |-- integrations/
    |   |-- activitywatch-browser-source/{index.ts,index.test.ts}
    |   `-- openviking-store/{index.ts,index.test.ts}
    `-- modules/
        |-- memory-events-cache/{index.ts,index.test.ts}
        `-- memory-review/{index.ts,index.test.ts,normalize-memory-items.ts}
```

Agentic completion. The replayed tree is already executable. Completion adds only the missing integration context needed to exercise the recovered task, including a memory event, an adapter, a React context, and runtime logs, while retaining the replayed source and tests. The generated `package-lock.json` is omitted.

ADDED **selected root-filesystem paths**

/

```
|-- .gitignore
|-- mirrorbrain/memory-events/
|   `-- browser:101.json
|-- src/
|   |-- apps/mirrorbrain-web-react/
|   |   |-- .gitignore
|   |   `-- src/contexts/MirrorBrainContext.tsx
|   `-- integrations/openviking-store/index.ts
`-- tmp/
    |-- mirrorbrain-debug.log
    `-- mirrorbrain-dev-new.log
```

Recovered intent. Intent Recovery reads the user request and trajectory evidence, rather than the completion model's additions, to produce the following task contract:

> Improve the MirrorBrain React interface by applying a polished, consistent, and accessible design across main navigation, subtabs, draft tabs, pagination controls, candidate cards, and action buttons, while preserving existing workflows and component behavior.

The source interaction realizes this contract through consistent active states, focus-visible styling, keyboard navigation, ARIA semantics, loading behavior, and a focused component test.

Generated queries. Single-WS treats the completed tree as a fresh execution context and does not copy the source solution. The resulting requests target observable interfaces already present in the project:

1. **Shell-event normalization.** Extend `normalize-memory-items.ts` so shell events already captured and rendered by the system can participate in daily review without regressing browser events.
2. **URL classification.** Replace substring-based development server detection with structured URL handling and distinguish GitHub Actions pages from ordinary repository pages.
3. **Client error handling.** Route five mutation methods in `api/client.ts` through consistent status-aware JSON handling, including descriptive behavior for non-JSON failures.
4. **Response schema alignment.** Extend the existing Fastify candidate-memory schema with `role` and `contribution` so React review components receive the required evidence metadata.

5. **Cache-source reconciliation.** Merge workspace and OpenViking events at cache initialization using existing ID and URL-signature deduplication, preserving reverse-chronological order.

F.2 Cross-WS Synthesis

This example exercises the directional dependency check in Cross-WS. Two C RSA toolkits share the same domain, but only one contains the decryption path needed for a complete round trip. The matcher therefore designates the incomplete toolkit as the target and the complete one as the evidence source. Table 9 lists the resulting synthesis stages.

Table 9: Evidence and checks used to construct the cross-workspace RSA task.

Stage and output	Evidence
Relation mining	Independently generated profiles classify both workspaces as C RSA toolkits; TF–IDF retrieval proposes the pair, and the relation judge records the directional capability gap "RSA decryption".
Pair packaging	The target is mounted writable at `/app/workspaces/repo1`; the reference is mounted read-only at `/app/workspaces/repo0`.
Query generation	The generator reads the actual files of both workspaces, confirms that the capability is implemented in the reference and absent from the target, and generates a task grounded in facts found only in the reference.
Verifier generation	An in-container agent authors the test suite and executes it on the initial workspace. New-capability tests fail while the preservation tests and the reference-unchanged check pass, satisfying the red-check requirement.
Solution rollout	The teacher solves the task; all clean checks pass, and the trajectory is retained in the pass-only corpus.

Mounted pair. The paired layout exposes a writable solver side and a read-only reference side. Their key files and input/output artifacts are shown below:

SOLVER **read-only reference and writable target · key files**

/app/workspaces/

```
|-- repo0/
|   |-- rsa_keygen.c
|   |-- rsa_encrypt.c
|   |-- rsa_decrypt.c
|   |-- rsa_sign.c
|   |-- input/
|   |   `-- plaintext.txt
|   `-- output/
|       `-- decrypted.txt
`-- repo1/
    |-- rsa_keygen.c
    |-- rsa_encrypt.c
    |-- rsa_sign.c
    |-- input/
    |   `-- plaintext.txt
    |-- output/
    |   `-- decrypted.txt
    `-- task_file/
        |-- input/
        |   `-- plaintext.txt
        `-- output/
```

Grounded capability gap. The target can generate keys, encrypt, and sign, but it has no implementation for the reverse operation. Its existing encryptor fixes conventions that the new program

must match, while the reference supplies the private-key path, decrypt invocation, padding, and file-location details. The generated request hides those reference facts and names only the mounted root, forcing the solver to inspect the evidence.

Selected request (abridged).

> The RSA cryptography toolkit at `/app/workspaces/repo1` currently supports key generation, encryption, and signing, but is missing the decryption program needed to complete the encrypt/decrypt round-trip. Create a new file `rsa_decrypt.c` in the root of `/app/workspaces/repo1` that reads an encrypted binary file and a private key, then decrypts the ciphertext back to plaintext. The program must integrate seamlessly with the existing toolkit: it should work with keys produced by the existing `rsa_keygen.c` and ciphertext produced by the existing `rsa_encrypt.c`, and it must follow the coding conventions, error-handling style, OpenSSL API usage patterns, and output path conventions already established by the other programs. A reference implementation exists at `/app/workspaces/repo0`: consult it to determine the exact OpenSSL API calls, padding mode, PEM reading function, file paths, command-line interface, and resource cleanup pattern required. When run after `rsa_keygen` and `rsa_encrypt`, the program must produce an output file whose contents are byte-for-byte identical to the original plaintext input.

What the verifier checks. The verifier checks compilation, the new executable, command-line behavior, file paths, padding, OpenSSL calls, and an end-to-end encrypt/decrypt round trip. Separate checks protect the existing target programs and confirm that the reference mount is unchanged. The teacher trajectory passes the complete suite.

F.3 Multi-Round User Queries

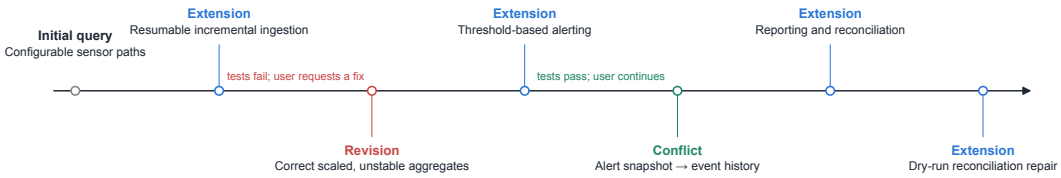

Figure 4: **Verifier-guided multi-round continuation.** Each state transition is checked privately before the user agent selects the next solver-visible request.

Table 10: Observed interaction styles among retained Multi-Round records; the generator does not enforce a target mixture.

Interaction style	Share
Feature extension	62.7%
Feature revision	29.3%
Feature conflict	8.0%

Table 11 shows how the private verifier outcome changes the next user-visible request while the workspace persists.

Table 11: Abridged solver-visible requests from the sensor-ingestion session. Bracketed ellipses indicate omitted text; a revision repairs the active contract, whereas a conflict intentionally replaces it.

Round	User-request excerpt
Initial request	"Working in the project root `/app`, the entrypoint `sensor_processor.py` hard-codes both its input path ... and its output path ..., which makes the script impossible to run against alternate data without editing source. Make it possible to invoke the script as `python sensor_processor.py <input_csv> <output_json>` so that the two positional arguments override the input and output locations."
1: Extension	"The path override work is solid, thanks. Now I need this to stop being a one-shot script, because in production my `sensor_data.csv` keeps growing and I don't want to re-crunch the entire file every time I get a new batch of readings. [...] Please add a resumable/incremental ingestion mode backed by a small persisted state store."
2: Revision	"The row-counting side of the incremental mode looks right — `readings_count` comes out at 150 per sensor on a fresh run and rises to exactly 154 after I append four rows, so old rows aren't being double-counted. But the actual aggregate values are wrong and unstable, and it's making the output useless for me. [...] It looks like each reading is being scaled down before it's aggregated."
3: Extension	"The aggregates are solid and stable now — thanks. The next thing I actually need for production is alerting, because right now I have to eyeball `processed_results.json` to notice when a sensor drifts out of its safe range. [...] Please add a threshold rules engine in a new `rules.py`, driven by a JSON rules config."
4: Conflict	"The alerting engine is doing its job now, but I've hit a real production gap: I've decided to change how `alerts.json` works. Right now it's overwritten every run holding only the new alerts for that run, which means when a sensor recovers the alert just silently vanishes and I have no record it ever fired or when it cleared. [...] Please turn `alerts.json` into an append-only history of transition events."
5: Extension	"The event history is finally durable, but right now `alerts.json` and `alerts.jsonl` are write-only — nothing ever reads them back, so when I come in Monday morning I still have to eyeball a growing history file to answer basic questions. I need a reporting command that turns that trail into an operational summary."
6: Extension	"The reconcile check has already caught me out once — it correctly flagged that `state.json` and the `alerts.json` history disagreed about what was open — but all it can do is tell me there's drift and exit non-zero. [...] I need reconcile to be able to actually fix the drift, not just report it. Please add a repair capability on top of the existing reconcile logic."

Failure-driven feature revision. The incremental mode initially introduces nondeterministic scaling. The private verifier detects the observable discrepancy, and the user agent requests a repair while keeping the active contract and hidden tests out of the solver conversation:

> The row counts are right, but the aggregate values are wrong and unstable. The averages, minima, and maxima are about half of the readings in the CSV, and rerunning unchanged input changes the result. Please make the aggregates use the real readings and remain deterministic for a fresh run, reset, and no-op rerun, while preserving the record shape, rounding, ordering, and incremental counts.

After the scaling defect is removed, the next verifier check passes and the user agent advances to threshold-based alerting. Test code and raw verifier results remain private throughout.

Table 12: Comparison of representative environment and training-data construction methods. Strategy summarizes the primary mechanism used to construct environments or training tasks. Env. counts reusable source repositories/workspaces or independently constructed environments, and Tasks gives the reported task or training-instance count. Verif. indicates an executable task verifier, Multi-Round denotes extended multi-round queries, and Cross-WS cross-workspace synthesis.

Method	**Strategy**	**Env.**	**Tasks**	**Verif.**	*Multi-Round*	*Cross-WS*
Endless Term. (Gandhi et al., 2026)	task-conditioned	3,255	3,255	✓	✗	✗
TMax (Ivison et al., 2026)	combinatorial sampling	14.6k	14.6k	✓	✗	✗
CLI-Gym (Lin et al., 2026)	env. perturbation	29	1,655	✓	✗	✗
CLI-Universe (Hua et al., 2026)	taxonomy-guided	6k	6k	✓	✗	✗
SkillSynth (Fan et al., 2026)	skill graph-guided	3,560	3,560	✓	✗	✗
OpenThinker-Agent (Raoof et al., 2026)	multi-source curation	–	100k	✓	✗	✗
RST (Li et al., 2026b)	recursive evolution	37.5k	37.5k	✓	✗	✗
CalibForge (Meng et al., 2026)	adversarial calibration	5,431	5,431	✓	✗	✗
REPO (ours)	trajectory reconstruction	37.3k	32.0k	✓	✓	✓

Controlled requirement conflict. Once alerting is working, the user agent invokes the explicit requirement-change policy to replace the per-run JSON snapshot with a durable event history. This is a product decision, not a correction to a failed implementation:

> I have decided to change how `alerts.json` works. It should no longer be a per-run snapshot: make it an append-only history of firing and resolved transitions, so a recovered alert remains recorded. Keep exit code 2 for a new firing and 0 for a lone resolution, and preserve the first-observed timestamp and severity with every event.

The replacement contract remains active for the remaining rounds, which then add reporting, reconciliation, and repair capabilities.

G Environment Construction Comparison

Table 12 compares representative approaches to constructing executable terminal environments and training tasks. Counts use the definitions reported by each work and are included for orientation rather than as a directly normalized benchmark.

H Reconstruction and Supplementary Ablations

This section probes the parts of REPO that are too detailed for the main comparison: state recovery, corpus composition, verifier filtering, data-budget allocation, and transfer from SWE traces. The values below use the same evaluation definitions as the main study.

Table 13: Task-sufficiency rates for replayed and completed workspaces, reported separately for Terminal and SWE source pools.

Pool	**# Environments**	**Sufficient (%)**	
		Post-replay	**Post-completion**
Terminal	38,294	40.2	**93.5**
SWE	1,900	20.1	**77.1**

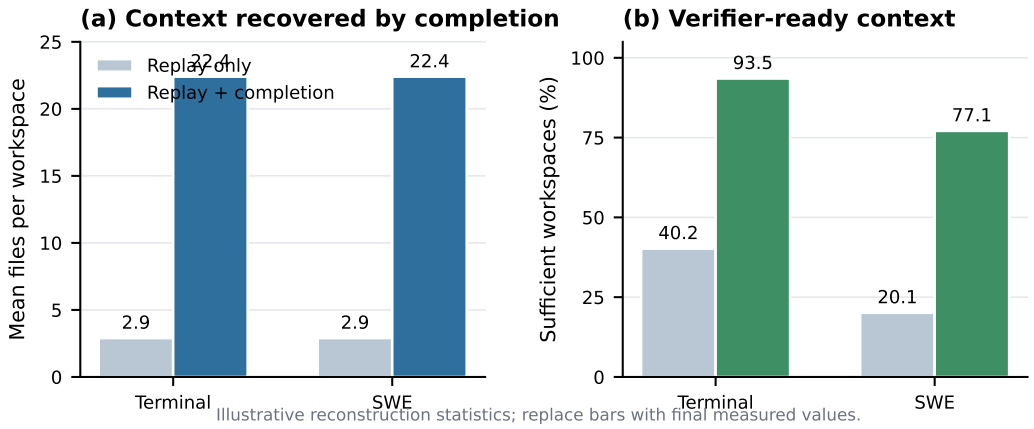

Figure 5: Effect of completion on recovered context and task sufficiency. The panels show file-level context and verifier judgments; the displayed census values remain provisional.

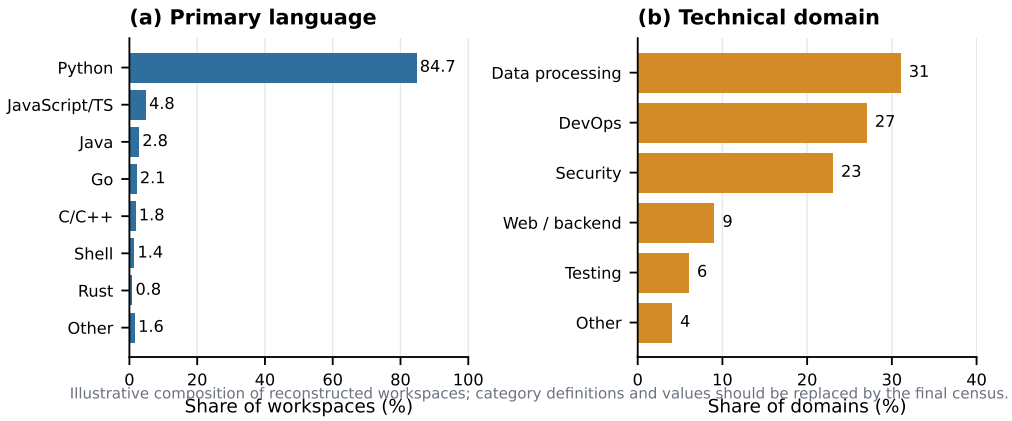

Figure 6: Illustrative language and domain composition of the reconstructed workspace pool. Rare categories are aggregated, and the percentages must be replaced by the final census.

Table 14: Contribution of verifier-based trajectory selection under a fixed launched-task pool for Single-WS and Cross-WS.

Variant	Records	Terminus2-XML
Single-WS w/o verifier	35.1k	$56.0_{3.3}$
Single-WS w/ verifier	25.4k	$\mathbf{56.4}_{2.6}$
Cross-WS w/o verifier	7.1k	$53.2_{2.1}$
Cross-WS w/ verifier	3.5k	$\mathbf{55.4}_{2.7}$

Table 15: Trajectory size and teacher success for the two workspace-querying modes under a common generation setup.

Variant	Turns		Tool calls		Tokens / record		Teacher pass@1
	Median	Mean	Median	Mean	Median	Mean	
Single-WS	14	15.1	20	21.2	30.4k	32.9k	72.3%
Cross-WS	23	25.3	38	39.7	46.5k	51.9k	49.2%

Table 16: Pilot category-level change from adding Cross-WS records to Single-WS training. The change is measured in Terminal-Bench 2.1 points; n is the number of tasks in the category. Final repeated-run estimates should replace these pilot values before submission.

Category	Change (points)	n
Software engineering	+2.2	26
Model training	+15.0	5
Debugging	+10.0	4
Data processing	−2.1	8
DevOps	−2.5	6
Security	+1.4	7

Table 17: Matched-record comparison of three ways to expand the Single-WS corpus: new environments, additional queries, or additional teacher solutions.

Variant	Envs	Queries / env	Solutions / query	Records	Terminus2-XML
Base pool	17,558	1	1	17.6k	$53.2_{4.1}$
Environment expansion	35,116	1	1	35.1k	$\mathbf{56.0}_{3.3}$
Query expansion	17,558	2	1	35.1k	$53.8_{3.3}$
Solution expansion	17,558	1	2	35.1k	$53.9_{2.4}$

Table 18: Transfer of Intent Recovery data from SWE workspaces to Terminal-Bench 2.1 under two evaluation scaffolds; estimates are provisional.

Variant	Data size	Claude Code	Terminus2-XML	Avg.
Qwen3.5-27B	-	$47.8_{2.0}$	$46.2_{3.9}$	47.0
Intent Recovery	10.3k	$\mathbf{50.6}_{0.9}$	$\mathbf{49.4}_{1.6}$	**50.0**

H.1 Supplementary Experimental Analyses

The analyses below isolate data-quality and scaling effects while keeping the main benchmark definitions and evaluation protocols fixed.

H.1.1 Role of Verifier Filtering

We first test whether verifier selection improves supervision enough to justify discarding failed teacher rollouts. For each query family, the comparison uses the full launched pool and its verifier-passed subset.

Selection is most useful for difficult cross-workspace tasks. Single-WS reaches 56.4 with the passed subset versus 56.0 when failures are retained, despite using fewer records. For Cross-WS,

retaining failed rollouts lowers the score to 53.2, while the passed subset reaches 55.4 with less than half as many records. The corresponding Multi-Round comparison also drops when round-level verification is removed (Table 20).

H.1.2 Expansion-Axis Ablation

We next hold the record budget approximately fixed and vary which axis supplies the additional supervision. From a base of 17,558 Single-WS environments with one query and one teacher solution each, we double either the number of environments, the number of queries per environment, or the number of solutions per query. Each resulting corpus contains roughly 35k records and uses the same Terminal-Bench 2.1 setup, so the comparison isolates budget allocation.

Environment expansion gives the largest increase, from 53.2 to 56.0; query and solution expansion reach 53.8 and 53.9. A new workspace contributes an independent execution context, whereas additional requests or solutions on an existing workspace share most of the same state. This result supports allocating early data budget to reconstructing diverse environments.

H.1.3 Cross-domain Generalization

We also test whether the replay-and-query pipeline transfers beyond terminal-native sources. We apply Intent Recovery to SWE workspaces, train with the same recipe, and measure the resulting policy on Terminal-Bench 2.1. Using Intent Recovery alone keeps the transfer test separate from the effects of Single-WS, Cross-WS, and Multi-Round task synthesis.

Among 1,900 SWE repositories, 1,464 pass the sufficiency judgment. Their multiple source tasks produce approximately 10.3k Intent Recovery trajectories. Training on these records raises the average from 47.0 to 50.0, with consistent movement under Claude Code (47.8 $\rightarrow$ 50.6) and Terminus2-XML (46.2 $\rightarrow$ 49.4). The effect therefore is not specific to one evaluation scaffold, although this experiment only measures transfer from SWE into terminal tasks.

Table 15 shows that the cross-workspace condition changes both trajectory length and task difficulty. Relative to Single-WS, Cross-WS uses 1.6$\times$ the median turns, 1.9$\times$ the tool calls, and 1.5$\times$ the tokens per record; teacher pass@1 falls from 72.3% to 49.2%. The additional cost comes from reconciling evidence in two codebases before editing, rather than from simply increasing the number of records.

How Cross-WS expands breadth. Single-WS samples one new requirement from a single recovered project. Cross-WS adds a second codebase as read-only evidence while keeping the target writable, creating capability combinations that cannot be expressed by the target workspace alone.

Where cross-workspace data helps. The mixture reaches $58.4_{2.1}$ under Terminus2-XML versus $56.4_{2.6}$ for Single-WS. Table 16 shows the category breakdown: the largest software engineering group rises from 46.9 to 49.1, with larger changes in model training (+15.0) and debugging (+10.0). Because several categories contain only a few tasks, these per-category values are diagnostic rather than standalone statistical claims; the aggregate result is consistent with Cross-WS contributing supervision that is unavailable from isolated workspaces.

H.2 Additional Expansion Tables

Table 19: Terminal-Bench 2.1 comparison of Single-WS, Cross-WS, and the combined training mixture.

Variant	Data size	Terminus2-XML
Qwen3.5-27B	–	$46.2_{3.9}$
Single-WS	25.4k	$56.4_{2.6}$
Cross-WS	3.5k	$55.4_{2.7}$
Single-WS + Cross-WS	28.9k	$\mathbf{58.4}_{2.1}$

Table 20: EvoCode-Bench v2 comparison of persistent Multi-Round data and the effect of removing round-level verifier feedback.

Variant	Data size	MT@4 score	Case score
Single-WS	25.4k	18.4	$71.9_{3.5}$
Single-WS + Multi-Round	28.5k	**21.0**	$\mathbf{76.9}_{1.4}$
Single-WS + Multi-Round (w/o round verifier)	28.5k	18.8	$73.2_{2.6}$

H.3 Compact Expansion Summary

Table 21: Matched summary of the breadth and depth comparisons reported above.

Expansion	Comparison	Metric	Change
Breadth	Single-WS $\rightarrow$ Single-WS + Cross-WS	TB2.1	$56.4 \rightarrow 58.4$
Depth	Single-WS $\rightarrow$ Single-WS + Multi-Round	MT@4 / Case	$18.4 \rightarrow 21.0$ / $71.9 \rightarrow 76.9$

I SFT Dataset Statistics

Table 22 reports corpus size together with median interaction length. The Multi-Round records are longer because each one preserves a completed state and appends several user decisions to the same session.

Table 22: Final SFT corpus statistics after length filtering.

Dataset	Records	Turns	Tool calls	Tokens
Intent Recovery	35,809	12	17	36.1k
Single-WS	25,386	14	20	30.4k
Cross-WS	3,512	23	38	46.5k
Multi-Round	3,079	92	102	126.1k

J Evaluation Configurations

Table 23 lists the configurations used for REPO-27B and the five reproduced open-weight baselines in Table 1. These reproduced baselines use model-specific settings rather than a shared evaluation configuration. The listed settings apply to both Terminal-Bench 2.1 and EvoCode-Bench v2 unless noted otherwise. For TMax-27B, the listed configuration applies only to our EvoCode-Bench v2 reproduction; its Terminal-Bench scores and Vanillux2Agent scaffold follow the original report. Qwen3.5-27B and Qwen3.7-Max use the REPO-27B configuration, with xhigh effort enabled for Qwen3.7-Max. On EvoCode-Bench v2, TermiGen-32B instead uses Terminus2-JSON with an 81,920-token summarization threshold.

Table 23: Evaluation configurations used in our reproductions.

Setting	Nemotron-Terminal-32B	OpenThinker-Agent-32B	TermiGen-32B	TerminalTraj-32B	TMax-27B	Terminal-Universe-27B
Architecture	`Qwen3ForCausalLM`	`Qwen3ForCausalLM`	`Qwen2ForCausalLM`	`Qwen2ForCausalLM`	`Qwen3_5ForConditionalGeneration`	`Qwen3_5ForConditionalGeneration`
Scaffold	Terminus2	Terminus2	BashAgent	Terminus2	Terminus2	Terminus2
Parser	JSON	JSON	Text JSON	JSON	XML	XML
Context length	40,960	32,768	131,072	32,768	262,144	262,144
Turn max tokens	8,192	8,192	8,192	8,192	65,536	65,536
Summ. threshold	32,960	28,672	disabled	24,768	176k	176k
Temperature	0.6	0.6	0.6	0.7	0.7	1.0
Top-p / top-k	0.95 / 20	0.95 / 20	0.8 / 20	0.8 / 20	0.95 / 20	0.95 / 20
Interleaved thinking	✗	✗	✗	✗	✓	✓
Thinking enabled	✓	✓	✗	✗	✓	✓
Shared across all reproductions						
TB wall clock	4 h					
EvoCode wall clock	10 h					
TB runs	6					
EvoCode runs	4					