RECURSIVE SYNTHESIS FOR LONG-HORIZON TERMINAL TASKS

Anonymous authors
Paper under double-blind review

ABSTRACT

High-quality long-horizon training data for terminal agents are costly to produce because the instruction, environment, reference solution, and verifier must remain aligned. Existing synthesis pipelines typically generate each task independently. We study how accepted executable tasks can be reused across rounds while checking for execution and specification errors. Recursive Synthetic Terminal Tasks (RST) applies this process to complete executable task bundles. In each round, RST extends the executable workflow of an accepted seed task, realigns the environment, verifier, and public instruction, validates the resulting candidate in a fresh sandbox, and reuses accepted candidates as seeds for the next round. Across 15 rounds, RST produces 37,484 accepted tasks while maintaining stable validation rates and broad domain and rewrite-operator coverage. Meanwhile, the median reference solution grows from 67 to 374 lines, static command count grows from 40 to 245, and DeepSeek-V4-Pro pass@4 falls from 90% in R_1 (round 1) to 2.5% in R_{15} (round 15). Supervised fine-tuning (SFT) on successful rollouts improves Qwen3.5-27B and Qwen3.5-122B-A10B across Terminal-Bench 2, Terminal-Bench Hard, and Long-Horizon Terminal Bench, with gains of up to 10 percentage points. Each updated checkpoint generates rollouts for the next SFT stage. Verifier-based proximal policy optimization (PPO) further raises Qwen3.5-27B to 49.81%, 32.00%, and 22.07% on the three benchmarks, corresponding to relative gains of 20.9%, 41.2%, and 21.9% over the base model. These results show that accepted executable tasks can serve both as training data and as seeds for generating progressively harder tasks. Code and the complete Terminal-Bench Hard evaluation subset are available at `https://anonymous.4open.science/r/recursive-synthesis-terminal-tasks-005F`. Synthesized tasks, trajectories, and checkpoints will be released upon publication.

1 INTRODUCTION

Large language models (LLMs) interact with tools and environments over many steps (Yang et al., 2026; Li et al., 2026c; Wang et al., 2026). Interactive benchmarks evaluate these agents on web, computer-use, and long-horizon tasks (Du et al., 2026; Chowa et al., 2026; Jimenez et al., 2024; Liu et al., 2024; Mialon et al., 2024). Terminal agents must coordinate repository context, command-line interaction, and execution feedback over extended workflows (Yang et al., 2024; Merrill et al., 2026; Desai et al., 2026). Training this behavior requires executable tasks that support long-horizon interaction (Team, 2026; Li et al., 2026e).

High-quality terminal tasks remain difficult to obtain at scale. Many existing tasks provide limited post-training signal for strong models, while new long-horizon tasks require a runnable workspace, a fair public instruction, meaningful execution feedback, and a reliable verifier (Vidgen et al., 2026; Jimenez et al., 2024; Merrill et al., 2026). Human-authored tasks and successful trajectories can cost hundreds to thousands of dollars per task (Yang et al., 2024; Wang et al., 2025b; Desai et al., 2026). Existing pipelines automate parts of task sourcing and environment construction (Jimenez et al., 2024; Zan et al., 2025; Wang et al., 2025a). However, extending a task's executable workflow requires coordinated changes to its reference solution, environment, verifier, and public instruction. A rewritten task may remain executable while its verifier checks requirements that the instruction and workspace do not expose. When accepted tasks are reused as seeds, these mismatches can propagate to subsequent rewrites. The central challenge is therefore to extend executable workflows

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.

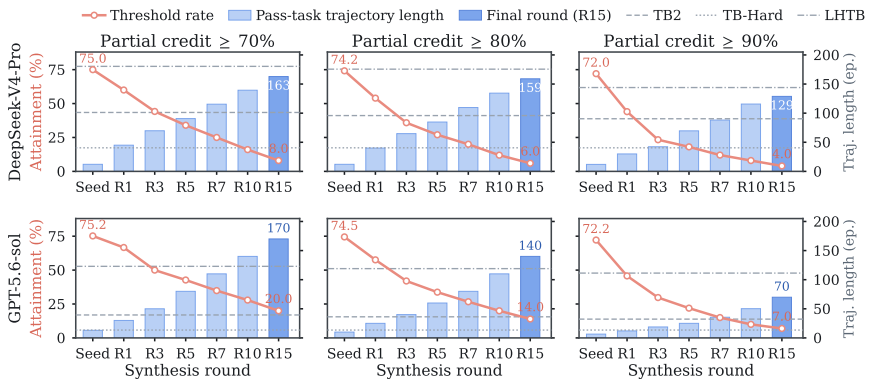

Figure 1: Partial-credit threshold-attainment rate and trajectory length across recursive synthesis rounds for DeepSeek-V4-Pro and GPT-5.6-sol at 70%, 80%, and 90% verifier-based partial-credit thresholds. Threshold-attainment rate is the share of evaluations reaching the specified partial-credit threshold. The trajectory length is measured in interaction steps for threshold-attaining episodes. Horizontal lines in the trajectory-length panels provide mean episode-length references from downstream terminal benchmarks.

across rounds while checking that each resulting task remains solvable and that its verifier evaluates the intended requirements.

Recursive Synthetic Terminal Tasks (RST) addresses this question. RST repeatedly reuses accepted executable tasks as seeds for later rounds. Within each rewrite, the reference solution provides an executable anchor: the system extends that workflow, updates the environment when needed, derives verifier checks from the resulting outcomes, and revises the public instruction last. The complete task is then executed and graded in a fresh sandbox before it can be reused. This process increases the executable work required by the task rather than merely lengthening its instruction. We use DeepSeek-V4-Pro (Xu et al., 2026) for synthesis and collect model-specific Qwen3.5 self-rollouts for training. In sequential SFT, each updated checkpoint generates trajectories on the next round's tasks. Successful trajectories then train that checkpoint. This feedback acts through SFT trajectory generation.

Across 15 rounds, RST produces 37,484 accepted tasks from 639 oracle-validated seeds. From R_1 to R_{15}, the median reference solution grows by 5.6× and static command count by 6.1×, while instruction length grows by only 1.4×. Under a fixed solver configuration, DeepSeek-V4-Pro pass@4 falls from 90% to 2.5%. Trajectories collected by rolling out each Qwen3.5 model on the synthesized tasks improve Qwen3.5-27B and Qwen3.5-122B-A10B through supervised fine-tuning on Terminal-Bench 2, Terminal-Bench Hard, and Long-Horizon Terminal Bench. Figure 1 previews the difficulty trend: across two fixed solvers, threshold attainment declines while successful trajectories generally require more interaction.

Our main contributions are:

1. **Recursive reuse of complete executable terminal tasks.** Accepted tasks seed subsequent synthesis rounds. Solution-first rewriting realigns the instruction, environment, reference solution, and verifier. Each candidate is then executed in a fresh sandbox. A controlled ablation evaluates solution-first rewriting and recursive seed reuse.
2. **Sustained task growth under explicit diversity control.** Across 15 rounds, the framework produces 37,484 accepted tasks without per-task human authoring. Generation API cost averages $0.05 per accepted task, excluding sandbox and training compute. Validation rates remain stable as structural complexity and fixed-solver difficulty increase, with broad domain and operator coverage under explicit selection constraints.
3. **Downstream training utility.** Self-collected trajectories improve both Qwen3.5 model scales across all three benchmarks, with gains reaching 10 percentage points. Comparisons under a fixed training-token budget and verifier-based PPO provide complementary evidence that the synthesized tasks offer useful supervision.

2 Related Work

Synthetic tasks and verifiable environments. Recent systems differ in what they synthesize. Some generate instructions or agent trajectories, while others construct executable tasks or environments for tool use, coding, web interaction, and terminal agents (Zhou et al., 2025; Xia et al., 2026; Zhu et al., 2025; Wu et al., 2026a; Ren et al., 2026; Kulikov et al., 2026; Li et al., 2026a; Song et al., 2026; Fang et al., 2025; Wang et al., 2025c). Endless Terminals and LiteCoder-Terminal study executable environments for supervised or reinforcement learning, while CLI-Universe, SETA, TermiGen, and ScaleEnv emphasize scalable task or environment construction (Gandhi et al., 2026; Peng et al., 2026; Hua et al., 2026; Shen et al., 2026; Zhu et al., 2026; Tu et al., 2026). RST recursively extends executable terminal tasks by updating each task's public instruction, environment, reference solution, and verifier to describe and evaluate the same extended workflow. Each candidate is validated in a fresh sandbox before it can serve as a seed for the next round. Appendix B provides a broader dataset comparison.

Recursive synthesis and self-improvement. Self-training, self-play, self-rewarding, and verifier-guided methods generate examples and retain those that satisfy a feedback signal (Zelikman et al., 2022; Li et al., 2026d; He et al., 2025; Singh et al., 2024; Chen et al., 2024; Yuan et al., 2024; Hosseini et al., 2024; Li et al., 2026f;b; Huang et al., 2026). Such loops can become unstable when rewards are misspecified or filtering is weak (Zhao et al., 2025; Prasad et al., 2025; Fu et al., 2025). In terminal settings, SETA adapts environments for agent training, BenchEvolver performs solution-centric task rewriting, and TRACE synthesizes trajectories through test-time exploration and reproduced outcomes (Shen et al., 2026; Wu et al., 2026b; Guo et al., 2025). RST builds on validated tasks to add executable work across rounds. It extends each task's reference solution and updates the environment, verifier, and instruction accordingly, checking the resulting task before further reuse.

Terminal-agent benchmarks and long-horizon evaluation. Repository-level benchmarks such as SWE-Bench, Multi-SWE-Bench, SWE-Bench++, and SWE-Marathon evaluate software agents with executable tests (Jimenez et al., 2024; Zan et al., 2025; Wang et al., 2025a; Desai et al., 2026). Terminal-Bench and Harbor focus on sandboxed command-line workflows with verifier-based grading (Merrill et al., 2026; Team, 2026), while Long-Horizon Terminal Bench and TerminalWorld emphasize persistent terminal interaction and tasks requiring many dependent steps (Li et al., 2026e; Chu et al., 2026). Broader web, desktop, and general-agent suites similarly stress state, delayed feedback, and repeated tool use (Liu et al., 2024; Mialon et al., 2024; Zhou et al., 2024; Xie et al., 2024; Kwa et al., 2025). These benchmarks evaluate agent capabilities, while RST generates increasingly demanding executable tasks for agent training through recursive expansion and validation.

3 Method

3.1 Problem Formulation

A terminal task is an executable bundle $x = (I, E, S, V)$. Here, I is the public instruction, E is the initialized environment and runtime metadata, S is a reference solution, and V is a private verifier. Harbor executes the task in an isolated sandbox with Terminus-2 as the agent harness (Team, 2026). The agent observes I and the workspace created by E, while V evaluates the final workspace state.

Our goal is to generate harder terminal tasks by extending the executable workflows of existing tasks across rounds. A validated task provides a working environment, a reference solution, and a verifier that can serve as the starting point for further expansion. Each extension adds work to the reference solution and updates the environment, verifier, and public instruction to match. The resulting task is checked before reuse because inconsistencies can otherwise propagate to later rounds. Seed selection also limits concentration on a small set of task origins or rewrite types.

Let $\mathcal{R}_r$ be the tasks accepted in synthesis round r. A rewrite operator o specifies a type of workflow extension, such as schema migration, test repair, or log diagnosis. The set $\mathcal{O}(x)$ contains operators supported by the artifacts and tools in x, with outcomes that can be checked from observable state. RST constructs a candidate $x' = \mathcal{T}(x, o)$ through solution-first rewriting and admits it to the next

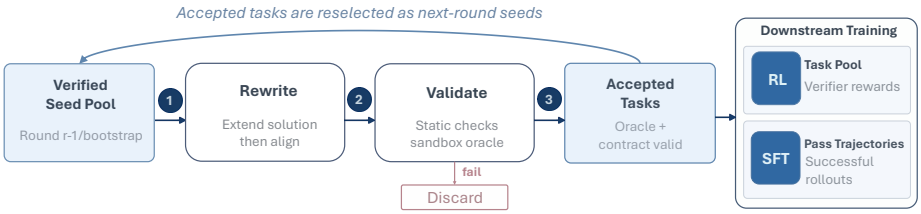

Figure 2: Overview of one RST round. Selected tasks undergo solution-first rewriting and automated validation. Accepted tasks seed the next synthesis round and form the reinforcement-learning pool, while successful agent rollouts provide supervised fine-tuning trajectories. In sequential SFT, each updated checkpoint generates the next round's rollouts.

round only after automated validation. Figure 2 shows this recursion and its connection to agent training.

3.2 Solution-First Bundle Realignment

Extending a task requires specifying which requirements remain unchanged and which are added. We capture these requirements in a task contract c that provides a shared specification for the reference solution, environment, verifier, and public instruction. The generator extends the reference solution under c and updates the other components to satisfy the same requirements. We summarize these dependencies as

$$\begin{aligned} c &= \mathrm{Contract}(x, o), \\ S' &= \mathrm{Extend}(S; x, c), \\ (I', E', V') &= \mathrm{Align}(x, c, S'), \qquad x' = (I', E', S', V'). \end{aligned} \tag{1}$$

Here, o is a rewrite operator from the taxonomy in Appendix D. $\mathrm{Contract}(x, o)$ specifies the retained and added requirements in c. $\mathrm{Extend}(S; x, c)$ produces the extended reference solution S'. $\mathrm{Align}(x, c, S')$ updates the public instruction I', environment E', and verifier V' to match S' and c. These components form the candidate task x', which is then validated.

The extended solution S' specifies how to complete the new task. We update the environment to provide the files, tools, and services needed to execute S'. We then update the verifier to check the required results and reject empty, stale, or hard-coded outputs. The public instruction states what the agent must accomplish, including requirements that cannot be inferred from the workspace. Before execution, a consistency check compares the instruction, environment, solution, and verifier with the requirements in c.

For example, the JSON-diff task in Appendix B.5 initially asks the agent to generate reports from fixed input files. A later version requires the agent to repair an invalid configuration before generating those reports. The reference solution therefore includes configuration repair, the workspace provides a configuration checker, and the instruction tells the agent to diagnose and repair the configuration. The verifier checks the required outputs of the extended task. Appendix Figure 21 summarizes the available rewrite types, whose definitions appear in Appendix D.

3.3 Validation and Recursive Reuse

Execution validity and specification consistency address different failure modes. Let $\mathrm{Exec}(S'; E')$ denote the outcome of running the reference solution in a fresh sandbox initialized from E'. Write $V'(\mathrm{Exec}(S'; E')) = \mathrm{pass}$ when the run succeeds under the constructed verifier. The executable acceptance condition is

$$\mathrm{OraclePass}(x') = \mathbf{1}\{V'(\mathrm{Exec}(S'; E')) = \mathrm{pass}\}\,. \tag{2}$$

A passing reference solution establishes execution success under V'. It does not establish that the public task and verifier express the same requirements. $\mathrm{ContractCheck}$ therefore applies automated checks intended to detect hidden or inconsistent requirements. Static checks also reject missing components, invalid metadata, private-verifier leakage, and changes too small to constitute a substantive

new task. Repairable failures receive bounded, file-restricted repair, followed by repeated static and sandbox checks.

With $\mathcal{T}$ denoting the rewrite procedure, including bounded repair when applicable, accepted candidates define the next pool:

$$\mathcal{R}_{r+1} = \{x' = \mathcal{T}(x, o) \mid x \in \mathrm{Select}(\mathcal{R}_r),\ o \in \mathcal{O}(x), \\ \mathrm{OraclePass}(x') = 1,\ \mathrm{ContractCheck}(x') = 1\}. \tag{3}$$

Equation 3 specifies the recursive acceptance rule. The checks remain imperfect proxies for semantic release readiness, which we assess separately through blinded human audit. Implementation and repair details appear in Appendix C and Appendix E.

Seed selection and population breadth. Seed selection balances task reliability and diversity. After removing duplicate or incomplete records, we rank tasks by file availability, environment cost, repair history, and available rollout statistics. We cap selection by bootstrap parent, category, rewrite family, and source group, then assign supported operators while balancing their usage. Appendix C details the selection rules and retained task records. Algorithm 1 summarizes a synthesis round.

Algorithm 1 One RST synthesis round

Input Accepted tasks $\mathcal{R}_r$
Output Accepted tasks $\mathcal{R}_{r+1}$

$\mathcal{R}_{r+1} \leftarrow \varnothing$
for $x = (I, E, S, V) \in \mathrm{Select}(\mathcal{R}_r)$ **do**
  $o \leftarrow \mathrm{ChooseOperator}(\mathcal{O}(x))$
  $c \leftarrow \mathrm{Contract}(x, o)$
  $S' \leftarrow \mathrm{Extend}(S; x, c)$
  $(I', E', V') \leftarrow \mathrm{Align}(x, c, S')$
  $x' \leftarrow (I', E', S', V')$
  $x' \leftarrow \mathrm{ValidateAndRepair}(x')$
  if $x' \neq \perp$ **then**
    $\mathcal{R}_{r+1} \leftarrow \mathcal{R}_{r+1} \cup \{x'\}$
  end if
end for
return $\mathcal{R}_{r+1}$

3.4 Connection to Agent Training

The synthesized tasks support iterative agent training. Starting from the base checkpoint π_0, each SFT stage updates π_{r-1} using its successful trajectories $\mathcal{D}_r$ on tasks $\mathcal{R}_r$,

$$\mathcal{D}_r = \mathrm{RetainSuccess}(\mathrm{Rollout}(\pi_{r-1}, \mathcal{R}_r)), \\ \pi_r = \mathrm{SFT}(\pi_{r-1}, \mathcal{D}_r). \tag{4}$$

The updated checkpoint generates the next stage's trajectories, coupling model updates with data collection. $\mathrm{RetainSuccess}$ follows the selection procedure in Appendix H; rollout statistics remain optional for seed selection. Accepted tasks also support verifier-based PPO. Section 4.5 explains the three-round SFT scope, and Appendix H gives training details.

4 Experiments

The experiments assess whether recursive task synthesis maintains task validity and diversity as difficulty increases, and whether the resulting tasks improve agent training. A controlled ablation isolates the effects of reference-solution-first rewriting and recursive seed reuse. Sequential SFT, comparisons under a fixed training-token budget, and verifier-based PPO evaluate downstream training utility.

Experimental setup. We initialize synthesis with 639 oracle-validated TerminalWorld tasks (Chu et al., 2026) and run 15 rounds. We assess candidate acceptance, structural growth, difficulty under

Table 1: Controlled 2×2 ablation. Round columns report validation pass rates. "Avg" averages rounds 1, 2, and 3, while "Post R_1 avg" averages rounds 2 and 3. The final column reports Terminal-Bench 2 after matched 3,000-example SFT. Values with $\pm$ are means and sample standard deviations over seeds 42, 43, and 44.

Group	R_1	R_2	R_3	Avg	Post-R_1 avg	TB2 after 3K SFT
Instruction-first, non-recursive	42.9±2.5	40.0±2.3	42.4±2.4	41.8	41.2	41.57±1.12
Instruction-first, recursive	42.9±2.5	60.0±3.5	58.3±3.2	53.7	59.2	42.32±1.72
Solution-first, non-recursive	66.7±1.2	64.3±1.0	66.7±1.1	65.9	65.5	43.07±0.65
Solution-first, recursive (ours)	**66.7±1.2**	**75.0±1.8**	**78.6±2.0**	**73.4**	**76.8**	**45.32±0.65**

fixed solver settings, task quality, diversity, and near-duplicate risk. To evaluate training utility, we perform SFT on Qwen3.5-27B and Qwen3.5-122B-A10B and verifier-based PPO on Qwen3.5-27B. Downstream evaluation covers Terminal-Bench 2 (TB2) (Merrill et al., 2026), Terminal-Bench Hard (TB-Hard), and Long-Horizon Terminal Bench (LHTB) (Li et al., 2026e). TB-Hard contains 100 tasks randomly sampled from TMax-15K (Ivison et al., 2026); the complete subset is included in the anonymous artifact. TB2 and TB-Hard measure the percentage of tasks solved, whereas LHTB measures mean verifier partial credit. Appendix H specifies decoding parameters, interaction horizons, timeouts, concurrency, and model-serving settings.

Corpus scale and benchmark separation. Synthesis produces 2,820 accepted tasks in R_1 and 37,484 across 15 rounds. Each seed submission counts as one attempt. Passed-task yield is $1{,}000N_{\text{pass}}/N_{\text{attempt}}$, whereas candidate pass rate is $N_{\text{pass}}/N_{\text{candidate}}$. In R_1, 5,112 attempts produce 3,637 candidates and 2,820 accepted tasks. DeepSeek-V4-Pro generation costs average $0.05 per accepted task, excluding sandbox and training compute. A textual overlap audit of 100 tasks from each of R_1, R_5, R_{10}, and R_{15} finds no shared normalized 13-token window with the three evaluation sets. Maximum 5-gram Jaccard similarity remains below 0.009, and unigram Jensen–Shannon divergence increases across rounds. Appendix B reports full results.

4.1 Ablating Generation and Reseeding

We isolate generation order and recursive seed reuse in a controlled 2×2 ablation conducted separately from the production run. Generation begins with either the public instruction or the reference solution. Seeds are drawn from either the original bootstrap pool or the tasks accepted in the preceding round. Each condition uses three generation runs with seeds 42, 43, and 44, producing 1,000 candidates per round in each run. The model, operator library, static checks, Daytona validation, and repair budget are held fixed. Within each generation order, both seed strategies share the same R_1 outcome because recursive reuse begins at R_2. To assess downstream effects under a matched sample budget, each condition uses 1,000 examples from each of the first three rounds for SFT, with independent training and evaluation runs using seeds 42, 43, and 44.

Both design choices improve validation rates (Table 1). Generating the reference solution first raises the three-round mean by 24.1 percentage points without recursive reuse and by 19.7 points with it. Recursive reuse raises the mean over rounds 2 and 3 by 18.0 points for instruction-first generation and 11.3 points for solution-first generation. Combining the two choices also yields the highest TB2 score under the matched 3,000-example SFT budget, reaching 45.32% compared with 41.57 to 43.07% for the other conditions.

4.2 Recursive Task Synthesis Results

Synthesis yield and candidate acceptance remain stable across the 15-round production run (Figure 3). Passed-task yield ranges from 498.2 to 572.2 accepted tasks per 1,000 seed submissions, with 551.6 in R_1 and 530.0 in R_{15}. Candidate pass rate ranges from 74.5% to 81.5%, with 77.5%

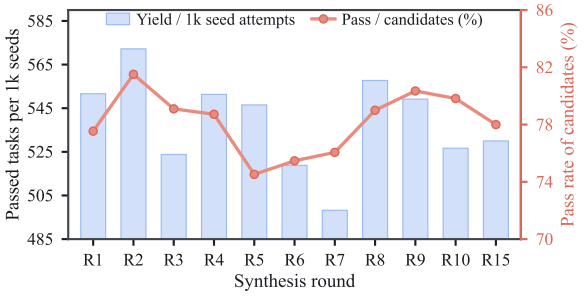

Figure 3: Synthesis validity across recursive rounds. Passed-task yield per 1,000 seed submissions and candidate pass rate remain stable through R_{15}.

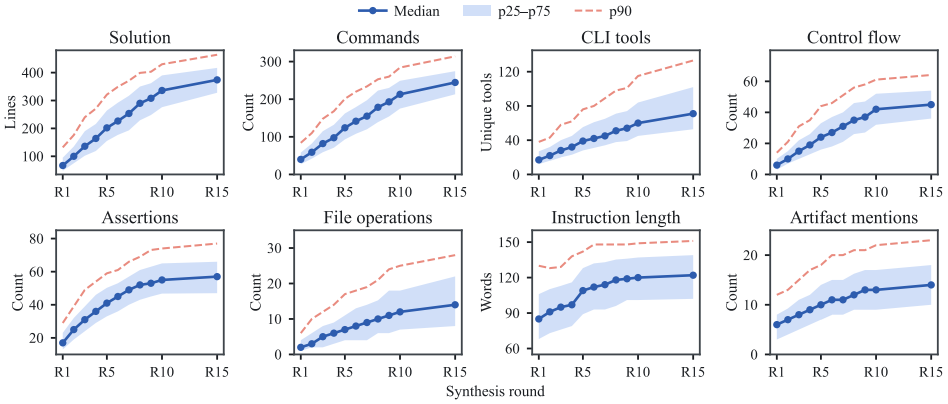

Figure 4: Task structure from R_1 to R_{15}. Executable complexity grows substantially, while instruction length grows more slowly.

in R_1 and 78.0% in R_{15}. These results show no sustained decline in automated acceptance over the evaluated rounds.

Recursive synthesis increases task complexity as measured by static structural indicators (Figure 4). From R_1 to R_{15}, the median reference-solution length increases from 67 to 374 lines, command count from 40 to 245, unique CLI tools from 17 to 71, control-flow operations from 6 to 45, file operations from 2 to 14, and verifier assertions from 17 to 57. In contrast, median instruction length increases from 85 to 122 words. The corresponding increases in upper quantiles indicate that structural growth extends beyond a small set of extreme tasks.

Quality audits assess whether later-round tasks remain useful and release-ready as their structural complexity increases. The LLM-rated utility proxy, which combines task value and release readiness, rises from 4.514 in R_1 to 4.761 in R_{15}. Release readiness remains above 4.6, decreasing from 4.796 to 4.629 over the same rounds. The blinded human audit finds that 86% of sampled R_{15} tasks are release-ready. Together, these results support continued task utility across rounds, while identifying residual quality defects among automatically accepted tasks. Appendix G and Appendix E.1 report the full quality and lexical audits.

4.3 Task Difficulty Analysis

To measure task difficulty beyond structural complexity, we evaluate DeepSeek-V4-Pro on 200 accepted tasks sampled from each round, with four attempts per task under fixed inference settings. Pass@4 measures the proportion of tasks completed in at least one attempt. For task i and attempt j, partial credit $PC_{i,j}$ is the fraction of equally weighted verifier checks satisfied. Task-level partial credit averages the four attempts, $PC_i = \frac{1}{4}\sum_{j=1}^{4} PC_{i,j}$.

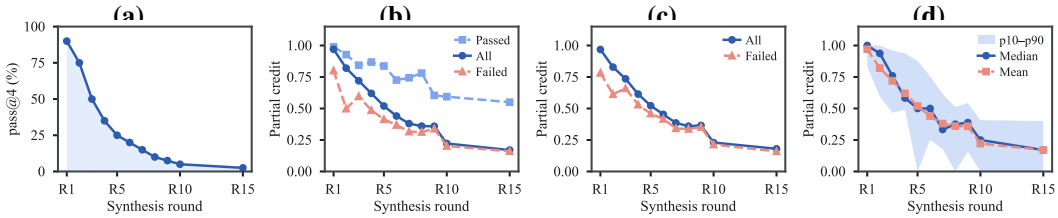

Figure 5: DeepSeek-V4-Pro performance on 200 randomly sampled tasks from each synthesis round. Each task is attempted four times. (a) The percentage of tasks solved at least once, which falls from 90% in R_1 to 2.5% in R_{15}. (b) The average fraction of verifier checks passed per task, shown for all tasks, solved tasks, and unsolved tasks. The overall average falls from 0.970 to 0.170. (c) Partial credit for individual attempts. (d) The distribution of task scores, shown by the mean, median, and 10th to 90th percentile range.

Figure 5 shows a monotonic decline in pass@4 from 90% in R_1 to 2.5% in R_{15}, a 36-fold reduction in full-task success. Mean task-level partial credit also decreases from 0.970 to 0.170. In R_{15}, the median is 0.170, with the 10th and 90th percentiles at 0 and 0.400, respectively. The concurrent decline in full-task success and partial credit shows that later-round tasks leave a larger fraction of verifier requirements unsatisfied under the same solver configuration. Appendix B provides near-miss and hard-failure breakdowns.

4.4 Diversity and Duplication Analysis

Increasing difficulty could coincide with a narrowing task distribution. Under the diversity-constrained reseeding mechanism, however, broad domain and rewrite-family coverage persists. Appendix Figures 13 and 14 show that the largest domain contains less than 25% of the tasks, the entropy-derived effective number of domains changes only from 11.22 in the seed pool to 11.09 in R_{15}, and the largest rewrite family contributes 24.9–28.9% of each round. These results show broad domain and rewrite-family coverage under the applied selection constraints.

Lineage and operator coverage narrow gradually but do not collapse. A lineage contains tasks descended from the same bootstrap task. Coverage of the 363 lineages represented in R_1 decreases from 98.3% in R_2 to 60.1% in R_{15}, while no lineage contributes more than 0.77% of any round. R_{15} still uses 36 of the 40 operators, and its most frequent operator accounts for 8.0% of tasks. Median within-round nearest-neighbor text similarity increases from 0.223 to 0.464, indicating that later tasks become more similar under this metric while remaining below 0.5 at the median. Appendix Figures 15, 16, and 18 provide the full distributions.

4.5 Supervised Fine-Tuning Results

To assess training utility, we retain 10,000 successful trajectories from each of R_1, R_2, and R_3 after deduplicating task records. The base checkpoint generates the R_1 rollouts and is then fine-tuned on the retained trajectories. For R_2 and R_3, the checkpoint from the preceding stage generates the new rollouts and is subsequently fine-tuned on that round's retained trajectories. We limit sequential SFT to the first three rounds because declining rollout success rates increase the cost of collecting successful trajectories in later rounds. The complete training and evaluation pipeline is repeated with seeds 42, 43, and 44.

Both model scales improve across successive training stages. Relative to their base models, Qwen3.5-27B gains 6.74, 5.66, and 4.34 percentage points on TB2, TB-Hard, and LHTB, respectively, while Qwen3.5-122B-A10B gains 5.62, 10.00, and 4.78 points. The gains emerge at different stages across benchmarks, suggesting that successive rounds contribute differently to downstream performance. Because each stage continues from the preceding checkpoint and generates the next round's rollouts, this experiment measures the *cumulative training utility* of successive rounds rather than the isolated contribution of any single round. Table 1 provides the controlled comparison for recursive reseeding. Appendix Figure 19 and Table 5 report the full results.

Table 2: Downstream performance, reported as percent mean ± sample standard deviation across seeds 42, 43, and 44. R3 SFT uses 10,000 trajectories per round, matched RST SFT uses 11,531 pooled R_1–R_3 trajectories at a fixed token budget, and PPO trains for 70 steps on the full task pool. For base models, we report three independent evaluation runs without post-training.

Model	Training route	TB2	TB-Hard	LHTB
Qwen3.5-27B	Base	41.20 ± 1.72	22.67 ± 2.52	18.10 ± 0.89
Qwen3.5-27B	R3 SFT	47.94 ± 2.34	28.33 ± 0.58	22.44 ± 0.40
Qwen3.5-27B	Matched RST SFT	46.82 ± 1.30	27.67 ± 1.53	21.90 ± 1.00
Qwen3.5-27B	PPO	$\mathbf{49.81 \pm 1.72}$	$\mathbf{32.00 \pm 2.00}$	22.07 ± 1.20
Qwen3.5-122B-A10B	Base	43.82 ± 2.97	20.00 ± 1.00	18.85 ± 1.16
Qwen3.5-122B-A10B	R3 SFT	49.44 ± 1.12	30.00 ± 1.73	$\mathbf{23.63 \pm 2.76}$

4.6 Comparison with Public Terminal-Agent SFT Data

To compare training utility under matched supervision, we evaluate RST against five public terminal-agent SFT datasets, including OpenThoughts-Agent-v1-SFT (Raoof et al., 2026), TMax-SFT (Ivison et al., 2026), LiteCoder-Terminal-SFT (Peng et al., 2026), Nemotron-Terminal-Corpus (skill-based) (Pi et al., 2026), and SETA-SFT (Shen et al., 2026). The RST condition uses 11,531 unique Qwen3.5-27B trajectories randomly sampled from pooled R_1–R_3 data. All conditions use exactly 36,363,275 assistant tokens for SFT, with the base checkpoint, preprocessing, optimization, and evaluation settings held fixed.

Under this matched token budget, RST achieves 46.82% on TB2, 27.67% on TB-Hard, and 21.90% on LHTB. These scores exceed the strongest public baseline by 4.12, 3.34, and 1.90 percentage points, respectively. RST also reaches the full training budget without repeating trajectories. The consistent gains indicate higher training utility per token across the three benchmarks, including long-horizon terminal tasks. Appendix Table 6 reports the full results.

4.7 Terminal-Agent Reinforcement Learning

We further evaluate whether the accepted task pool supports verifier-based reinforcement learning. Starting from the base policy, we train Qwen3.5-27B for 70 PPO steps with seeds 42, 43, and 44. The critic is initialized from a terminal-agent checkpoint trained on TMax after excluding all 100 TB-Hard evaluation tasks. Across the three runs, scores range from 48.31% to 51.69% on TB2, from 30.00% to 34.00% on TB-Hard, and from 20.90% to 23.30% on LHTB.

The corresponding mean scores are 49.81%, 32.00%, and 22.07%, improving over the base model by 8.61, 9.33, and 3.97 percentage points, respectively. The largest relative gain is 41.16% on TB-Hard. PPO achieves the strongest Qwen3.5-27B results on TB2 and TB-Hard, while remaining slightly below sequential SFT on LHTB. This smaller LHTB gain may reflect the mismatch between the 120-turn PPO rollout limit and the longer evaluation horizon. Appendix Figure 20 presents the training dynamics, and Table 7 reports the full comparison.

5 Conclusion

We introduced RST, which recursively extends validated terminal tasks and realigns their reference solution, environment, verifier, and public instruction before fresh-sandbox validation. Across 15 rounds, RST produces 37,484 accepted tasks with stable validation rates, while median solution length increases from 67 to 374 lines and fixed-solver performance declines. Training on the resulting trajectories improves both Qwen3.5 model scales. Under a matched token budget, RST outperforms the strongest of five public SFT baselines across all three benchmarks, while PPO further improves Qwen3.5-27B. Controlled ablations support the benefits of reference-solution-first rewriting and recursive seed reuse. The current results remain limited by bootstrap-task coverage and solver-dependent difficulty measurements.

AI USE STATEMENT

We used generative AI tools for task synthesis, quality review, and manuscript preparation. All AI-assisted outputs were validated through automated checks and human review. We take full responsibility for the final content, claims, experiments, and artifacts.

ETHICS STATEMENT

This work studies synthetic terminal-agent tasks and does not involve human-subject experiments. The released task environments are intended for research on executable agent training and evaluation, but stronger terminal agents can have dual-use implications because they may automate software and system operations. We mitigate these risks by using sandboxed execution, verifier-based task definitions, and generated task bundles that should be run only in isolated environments. The dataset construction process is designed to avoid private credentials, personal data, and hidden requirements in public task instructions.

REPRODUCIBILITY STATEMENT

The appendix documents the recursive synthesis implementation, rewrite operators, filtering and repair policy, prompt templates, and training and evaluation configuration. The anonymous artifact at `https://anonymous.4open.science/r/recursive-synthesis-terminal-tasks-005F` provides the code and complete Terminal-Bench Hard evaluation subset. The released artifacts will additionally include synthesized task bundles, sampled trajectories, trained checkpoints, and manifests needed to trace tasks across synthesis rounds. The reported analyses are computed from recorded manifests and validation logs, and evaluations use fixed benchmark configurations for Terminal-Bench 2, Terminal-Bench Hard, and Long-Horizon Terminal Bench.

REFERENCES

Zixiang Chen, Yihe Deng, Huizhuo Yuan, Kaixuan Ji, and Quanquan Gu. Self-play fine-tuning converts weak language models to strong language models. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 6621–6642. PMLR, 21–27 Jul 2024. URL `https://proceedings.mlr.press/v235/chen24j.html`.

Sadia Sultana Chowa, Riasad Alvi, Subhey Sadi Rahman, Md Abdur Rahman, Mohaimenul Azam Khan Raiaan, Md Rafiqul Islam, Mukhtar Hussain, and Sami Azam. From language to action: a review of large language models as autonomous agents and tool users. *Artificial Intelligence Review*, 59(2):71, 2026.

Zhaoyang Chu, Jiarui Hu, Xingyu Jiang, Pengyu Zou, Han Li, Chao Peng, Peter O'Hearn, Earl T Barr, Mark Harman, Federica Sarro, et al. Terminalworld: Benchmarking agents on real-world terminal tasks. *arXiv preprint arXiv:2605.22535*, 2026.

Rishi Desai, Jesse Hu, Joan Cabezas, Neel Harsola, Pratyush Shukla, Roey Ben Chaim, Adnan El Assadi, Omkaar Mukund Kamath, Fenil Faldu, Prannay Hebbar, et al. Swe-marathon: Can agents autonomously complete ultra-long-horizon software work? *arXiv preprint arXiv:2606.07682*, 2026.

Shangheng Du, Jiabao Zhao, Jinxin Shi, Zhentao Xie, Xin Jiang, Yanhong Bai, and Liang He. A survey on the optimization of large language model-based agents. *ACM Computing Surveys*, 58 (9):1–37, 2026.

Tianqing Fang, Hongming Zhang, Zhisong Zhang, Kaixin Ma, Wenhao Yu, Haitao Mi, and Dong Yu. WebEvolver: Enhancing web agent self-improvement with co-evolving world model. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 8959–8975, Suzhou, China, November 2025. Association for Computational Linguistics. ISBN 979-8-89176-332-6. doi: 10.18653/v1/2025.emnlp-main.454. URL `https://aclanthology.org/2025.emnlp-main.454/`.

Shi Fu, Yingjie Wang, Yuzhu Chen, Li Shen, and Dacheng Tao. Self-verification provably prevents model collapse in recursive synthetic training. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 36101–36154. Curran Associates, Inc., 2025. doi: 10.52202/085713-1213. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/3380e8116452e0efbf36f35d95e88c94-Paper-Conference.pdf.

Kanishk Gandhi, Shivam Garg, Noah D Goodman, and Dimitris Papailiopoulos. Endless terminals: Scaling rl environments for terminal agents. *arXiv preprint arXiv:2601.16443*, 2026.

Dadi Guo, Tianyi Zhou, Dongrui Liu, Chen Qian, Qihan Ren, Shuai Shao, Zhiyuan Fan, Yi R Fung, Kun Wang, Linfeng Zhang, et al. Towards self-evolving benchmarks: Synthesizing agent trajectories via test-time exploration under validate-by-reproduce paradigm. *arXiv preprint arXiv:2510.00415*, 2025.

Yicheng He, Chengsong Huang, Zongxia Li, Jiaxin Huang, and Yonghui Yang. Visplay: Self-evolving vision-language models from images. *arXiv preprint arXiv:2511.15661*, 2025.

Arian Hosseini, Xingdi Yuan, Nikolay Malkin, Aaron Courville, Alessandro Sordoni, and Rishabh Agarwal. V-STar: Training verifiers for self-taught reasoners. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=stmqBSW2dV.

Zhanbo Hua, Yifan Yao, Weihao Xie, Yongchi Zhao, Minghao Liu, Ruizhi Qiu, Zhewei Huang, Zun Wang, Yiyan Ji, Yunhai Ye, et al. Cli-universe: Towards verifiable task synthesis engine for terminal agents. *arXiv preprint arXiv:2606.22883*, 2026.

Chengsong Huang, Wenhao Yu, Xiaoyang Wang, Hongming Zhang, Zongxia Li, Ruosen Li, Jiaxin Huang, Haitao Mi, and Dong Yu. R-zero: Self-evolving reasoning llm from zero data. In *International Conference on Learning Representations*, volume 2026, pp. 130770–130790, 2026.

Hamish Ivison, Junjie Oscar Yin, Rulin Shao, Teng Xiao, Nathan Lambert, and Hannaneh Hajishirzi. Tmax: A simple recipe for terminal agents. *arXiv preprint arXiv:2606.23321*, 2026.

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 B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (eds.), *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/edac78c3e300629acfe6cbe9ca88fb84-Paper-Conference.pdf.

Ilia Kulikov, Chenxi Whitehouse, Tianhao Wu, Yixin Nie, Swarnadeep Saha, Eryk Helenowski, Weizhe Yuan, Olga Golovneva, Jack Lanchantin, Yoram Bachrach, et al. Autodata: An agentic data scientist to create high quality synthetic data. *arXiv preprint arXiv:2606.25996*, 2026.

Thomas Kwa, Ben West, Joel Becker, Amy Deng, Katharyn Garcia, Max Hasin, Sami Jawhar, Megan Kinniment, Nate Rush, Sydney Von Arx, Ryan Bloom, Thomas Broadley, Haoxing Du, Brian Goodrich, Nikola Jurkovic, Luke Miles, Seraphina Nix, Tao Lin, Neev Parikh, David Rein, Lucas Jun Koba Sato, Hjalmar Wijk, Daniel Ziegler, Elizabeth Barnes, and Lawrence Chan. Measuring ai ability to complete long software tasks. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 92213–92266. Curran Associates, Inc., 2025. doi: 10.52202/085713-3086. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/85069585133c4c168c865e65d72e9775-Paper-Conference.pdf.

Jiaxi Li, Ke Deng, Yun Wang, Jingyuan Huang, Yucheng Shi, Qiaoyu Tan, Jin Lu, and Ninghao Liu. Online skill learning for web agents via state-grounded dynamic retrieval. *arXiv preprint arXiv:2606.04391*, 2026a.

Jiaxi Li, Yucheng Shi, Xiao Huang, Jin Lu, and Ninghao Liu. Mits: Enhanced tree search reasoning for llms via pointwise mutual information. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pp. 288–300. Springer, 2026b.

Zhongzhi Li, Xuansheng Wu, Yijiang Li, Lijie Hu, and Ninghao Liu. Less is enough: Synthesizing diverse data in LLM feature space with sparse autoencoders. In *Forty-third International Conference on Machine Learning*, 2026c. URL https://openreview.net/forum?id=GVupwQ0578.

Zongxia Li, Hongyang Du, Chengsong Huang, Xiyang Wu, Lantao Yu, Yicheng He, Jing Xie, Xiaomin Wu, Zhichao Liu, Jiarui Zhang, et al. Mm-zero: Self-evolving multi-model vision language models from zero data. *arXiv preprint arXiv:2603.09206*, 2026d.

Zongxia Li, Zhongzhi Li, Yucheng Shi, Ruhan Wang, Junyao Yang, Zhichao Liu, Xiyang Wu, Anhao Li, Yue Yu, Ninghao Liu, et al. Long-horizon-terminal-bench: Testing the limits of agents on long-horizon terminal tasks with dense reward-based grading. *arXiv preprint arXiv:2607.08964*, 2026e.

Zongxia Li, Wenhao Yu, Chengsong Huang, Zhenwen Liang, Rui Liu, Fuxiao Liu, Jingxi Chen, Dian Yu, Jordan Boyd-Graber, Haitao Mi, and Dong Yu. Vision-sr1: Self-rewarding vision-language model via reasoning decomposition and multi-reward policy optimization. In C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (eds.), *International Conference on Learning Representations*, volume 2026, pp. 81753–81772, 2026f. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/8442d394d8266db6fdf92583a4783e8c-Paper-Conference.pdf.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. Agentbench: Evaluating llms as agents. In *International Conference on Learning Representations*, volume 2024, pp. 52989–53046, 2024.

Mike Merrill, Alexander Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *International Conference on Learning Representations*, volume 2026, pp. 40903–40986, 2026.

Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. Gaia: a benchmark for general ai assistants. In *International Conference on Learning Representations*, volume 2024, pp. 9025–9049, 2024.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with SWE-gym. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 47717–47737. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/pan25g.html.

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. *arXiv preprint arXiv:2605.29559*, 2026.

Renjie Pi, Grace Lam, Mohammad Shoeybi, Pooya Jannaty, Bryan Catanzaro, and Wei Ping. On data engineering for scaling llm terminal capabilities. *arXiv preprint arXiv:2602.21193*, 2026.

Archiki Prasad, Weizhe Yuan, Richard Yuanzhe Pang, Jing Xu, Maryam Fazel-Zarandi, Mohit Bansal, Sainbayar Sukhbaatar, Jason E Weston, and Jane Yu. Self-consistency preference optimization. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 49737–49751. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/prasad25a.html.

Negin Raoof, Richard Zhuang, Marianna Nezhurina, Etash Guha, Atula Tejaswi, Ryan Marten, Charlie F Ruan, Tyler Griggs, Alexander Glenn Shaw, Hritik Bansal, et al. Openthoughts-agent: Data recipes for agentic models. *arXiv preprint arXiv:2606.24855*, 2026.

Jincheng Ren, Siwei Wu, Yizhi Li, Kang Zhu, Shu Xu, Boyu Feng, Ruibin Yuan, Wei Zhang, Riza Batista-Navarro, Jian Yang, et al. A self-evolving framework for efficient terminal agents via observational context compression. *arXiv preprint arXiv:2604.19572*, 2026.

Qijia Shen, Zhiqi Huang, Vamsidhar Kamanuru, Aznaur Aliev, Jay Rainton, Ahmed Awelkair, Zhichen Zeng, Jiajun Li, Shi Dong, Yueming Yuan, et al. Seta: Scaling environments for terminal agents. *arXiv preprint arXiv:2607.10891*, 2026.

Avi Singh, John D Co-Reyes, Rishabh Agarwal, Ankesh Anand, Piyush Patil, Xavier Garcia, Peter J Liu, James Harrison, Jaehoon Lee, Kelvin Xu, Aaron T Parisi, Abhishek Kumar, Alexander A Alemi, Alex Rizkowsky, Azade Nova, Ben Adlam, Bernd Bohnet, Gamaleldin Fathy Elsayed, Hanie Sedghi, Igor Mordatch, Isabelle Simpson, Izzeddin Gur, Jasper Snoek, Jeffrey Pennington, Jiri Hron, Kathleen Kenealy, Kevin Swersky, Kshiteej Mahajan, Laura A Culp, Lechao

Xiao, Maxwell Bileschi, Noah Constant, Roman Novak, Rosanne Liu, Tris Warkentin, Yamini Bansal, Ethan Dyer, Behnam Neyshabur, Jascha Sohl-Dickstein, and Noah Fiedel. Beyond human data: Scaling self-training for problem-solving with language models. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856. URL https://openreview.net/forum?id=lNAyUngGFK. Expert Certification.

Xiaoshuai Song, Haofei Chang, Guanting Dong, Yutao Zhu, Ji-Rong Wen, and Zhicheng Dou. Envscaler: Scaling tool-interactive environments for llm agent via programmatic synthesis. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 8326–8357, 2026.

Harbor Framework Team. Harbor: A framework for evaluating and optimizing agents and models in container environments. *Zenodo*, 2026.

Dunwei Tu, Hongyan Hao, Hansi Yang, Yihao Chen, Yu Yang, Yueqing Sun, Xingchen Liu, Furao Shen, Qi GU, Hui Su, and Xunliang Cai. Scaleenv: Scaling environment synthesis from scratch for generalist interactive tool-use agent training. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=4GX4r1CYtD.

Bertie Vidgen, Austin Mann, Abby Fennelly, John Wright Stanly, Lucas Rothman, Marco Burstein, Julien Benchek, David Ostrofsky, Anirudh Ravichandran, Debnil Sur, et al. Apex-agents. *arXiv preprint arXiv:2601.14242*, 2026.

Lilin Wang, Lucas Ramalho, Alan Celestino, Phuc Anthony Pham, Yu Liu, Umang Kumar Sinha, Andres Portillo, Onassis Osunwa, and Gabriel Maduekwe. Swe-bench++: A framework for the scalable generation of software engineering benchmarks from open-source repositories. *arXiv preprint arXiv:2512.17419*, 2025a.

Ruhan Wang, Yucheng Shi, Zongxia Li, Zhongzhi Li, Yue Yu, Junyao Yang, Kishan Panaganti, Haitao Mi, Dongruo Zhou, et al. Harness handbook: Making evolving agent harnesses readable, navigable, and editable. *arXiv preprint arXiv:2607.13285*, 2026.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025b.

Yiming Wang, Da Yin, Yuedong Cui, Ruichen Zheng, Zhiqian Li, Zongyu Lin, Di Wu, Xueqing Wu, Chenchen Ye, Yu Zhou, et al. Llms as scalable, general-purpose simulators for evolving digital agent training. *arXiv preprint arXiv:2510.14969*, 2025c.

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. In *Forty-third International Conference on Machine Learning*, 2026a. URL https://openreview.net/forum?id=PeFSCRulgy.

Yangzhen Wu, Aaron J Li, Wenjie Ma, Li Cao, Ziheng Zhou, Mert Cemri, Shu Liu, Yuran Xiu, Chenxiao Yan, Haikun Zhao, et al. Benchevolver: Frontier task synthesis via solution-centric evolution. *arXiv preprint arXiv:2606.01286*, 2026b.

Peng Xia, Kaide Zeng, Jiaqi Liu, Can Qin, Fang Wu, Yiyang Zhou, Caiming Xiong, and Huaxiu Yao. Agent0: Unleashing self-evolving agents from zero data via tool-integrated reasoning. In *Third Conference on Language Modeling*, 2026. URL https://openreview.net/forum?id=40rA9OrNya.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094, 2024.

Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, et al. Deepseek-v4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026.

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, Qingwei Lin, and Daxin Jiang. Wizardlm: Empowering large pre-trained language models to follow complex instructions. In *International Conference on Learning Representations*, volume 2024, pp. 30745–30766, 2024.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering.

Advances in Neural Information Processing Systems, 37:50528–50652, 2024.

Junyao Yang, Yucheng Shi, Zongxia Li, Zhongzhi Li, Ruhan Wang, Xiangxin Zhou, Kishan Panaganti, Haitao Mi, and Leowei Liang. Stale but stable: Staleness-adaptive trust regions for stabilizing asynchronous reinforcement learning. *arXiv preprint arXiv:2607.18722*, 2026.

Weizhe Yuan, Richard Yuanzhe Pang, Kyunghyun Cho, Xian Li, Sainbayar Sukhbaatar, Jing Xu, and Jason E Weston. Self-rewarding language models. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 57905–57923. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/yuan24d.html.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Shulin Xin, Linhao Zhang, Qi Liu, Li Aoyan, Lu Chen, Xiaojian Zhong, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, RUI LONG, Ming Ding, and liang xiang. Multi-swe-bench: A multilingual benchmark for issue resolving. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference. Curran Associates, Inc., 2025. doi: 10.52202/085713-2111. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/5afa9cb1e917b898ad418216dc726fbd-Paper-Datasets_and_Benchmarks_Track.pdf.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. Star: Bootstrapping reasoning with reasoning. *Advances in Neural Information Processing Systems*, 35:15476–15488, 2022.

Zhiyuan Zeng, Hamish Ivison, Yiping Wang, Lifan Yuan, Shuyue Stella Li, Zhuorui Ye, Siting Li, Jacqueline He, Runlong Zhou, Tong Chen, Chenyang Zhao, Yulia Tsvetkov, Simon Shaolei Du, Natasha Jaques, Hao Peng, Pang Wei Koh, and Hannaneh Hajishirzi. RLVE: Scaling up reinforcement learning for language models with adaptive verifiable environments. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=DRqjMnDQGq.

Andrew Zhao, Yiran Wu, Tong Wu, Quentin Xu, Yang Yue, Matthieu Lin, Shenzhi Wang, Qingyun Wu, Zilong Zheng, and Gao Huang. Absolute zero: Reinforced self-play reasoning with zero data. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 105816–105879. Curran Associates, Inc., 2025. doi: 10.52202/085713-3534. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/9837dc00ff67d176373268ed48042d49-Paper-Conference.pdf.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. Webarena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*, volume 2024, pp. 15585–15606, 2024.

Yifei Zhou, Sergey Levine, Jason Weston, Xian Li, and Sainbayar Sukhbaatar. Self-challenging language model agents. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 113959–113991. Curran Associates, Inc., 2025. doi: 10.52202/085713-3801. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/a5a305fac88fb4ae40969cfec5eef48d-Paper-Conference.pdf.

Kaijie Zhu, Yuzhou Nie, Yijiang Li, Yiming Huang, Jialian Wu, Jiang Liu, Ximeng Sun, Zhenfei Yin, Lun Wang, Zicheng Liu, et al. Termigen: High-fidelity environment and robust trajectory synthesis for terminal agents. *arXiv preprint arXiv:2602.07274*, 2026.

Yiqi Zhu, Apurva Gandhi, and Graham Neubig. Training versatile coding agents in synthetic environments. *arXiv preprint arXiv:2512.12216*, 2025.

APPENDIX

This appendix provides methodological details, validation analyses, implementation records, prompt templates, and additional experiments that support the claims in the main paper. Sections A and B document the bootstrap data and supporting empirical evidence. Sections C–F describe the synthesis implementation, operator taxonomy, quality safeguards, and prompts. Sections G and H report the controlled task-quality comparison and the complete training and evaluation setup. The appendix is organized as follows.

- **A. Bootstrap Seed Pool Diversity.** Appendix A documents how the 639 oracle-validated bootstrap tasks are grouped into broader analysis domains. It gives the complete mapping from the original fine-grained TerminalWorld categories to the 19 domains used for diversity and coverage analyses.

- **B. Additional Experimental Figures and Tables.** Appendix B collects supporting empirical evidence referenced from the main text while preserving the original figure and table formats. Appendix B.1 reports direct textual overlap with Terminal-Bench 2, Terminal-Bench Hard, and Long-Horizon Terminal Bench. Appendix B.2 expands the synthesis pipeline and structural-growth diagnostics. Appendix B.3 details fixed-solver failure modes, domain stability, parent and operator coverage, novelty, and duplicate-risk measurements. Appendix B.4 provides round-by-round SFT results and the dataset comparison table. Appendix B.5 gives a concrete recursive lineage case study for a JSON-diff regression task.

- **C. Recursive Synthesis Implementation.** Appendix C describes the manifest-driven synthesis system, seed selection and materialization, file allowlist, operator assignment, transformation contracts, oracle validation, repair records, and lineage safeguards used across recursive rounds.

- **D. Operator Taxonomy.** Appendix D lists the 40 rewrite operators used by the synthesis pipeline and explains how the five operator families cover environment, build, data, configuration, and diagnostic sources of terminal-task difficulty.

- **E. Filters, Preflight, and Quality Control.** Appendix E details the generation-quality filter, static preflight checks, sandbox oracle validation, and bounded feedback-repair policy. Appendix E.1 further explains the instruction–verifier discoverability rules and contract-granularity variants used to prevent hidden requirements.

- **F. Synthesis Prompt Templates.** Appendix F reproduces the fixed prompt templates used by RST, with run-dependent seed files, operator cards, task contracts, and Daytona feedback replaced by labeled placeholders.

- **G. Task-Quality Comparison: Recursive versus Single-Pass Rewriting.** Appendix G reports model-based and blinded human audits of recursive synthesis rounds against instruction-first and solution-first single-pass rewriting baselines.

- **H. Training and Evaluation Implementation.** Appendix H records the SFT, PPO, model-serving, Harbor, Daytona, rollout, and benchmark-evaluation configurations used to reproduce the training and evaluation results.

A Bootstrap Seed Pool Diversity

Section 3.3 describes the selection mechanism applied to the 639 oracle-validated bootstrap tasks, whose high-level domain distribution is shown in Figure 14. We consolidate the original fine-grained category labels into broader domains for analysis. Table 3 provides the complete mapping between the original categories and the resulting analytical groupings.

B Additional Experimental Figures and Tables

This section collects supporting empirical evidence referenced from the main text.

B.1 Benchmark Text-Overlap Audit

The overlap audit measures direct textual similarity between task descriptions. For each audited round, we draw 100 accepted tasks using deterministic proportional stratification with seed 42. Instructions are NFKC-normalized, lowercased, whitespace-collapsed, and tokenized. We then check exact shared 13-token windows and nearest-neighbor 5-gram Jaccard similarity against all tasks in Terminal-Bench 2, Terminal-Bench Hard, and Long-Horizon Terminal Bench.

Table 4 shows no exact 13-token match with any task in the three benchmarks. The maximum 5-gram Jaccard similarity remains below 0.009 in every audited round. It decreases from 0.0081 in R_1 to 0.0028 in R_{10} and remains low at 0.0051 in R_{15}. Meanwhile, unigram Jensen–Shannon divergence (JSD) increases from R_1 to R_{15} for all three benchmarks: from 0.358 to 0.433 for Terminal-Bench 2, from 0.441 to 0.485 for Long-Horizon Terminal Bench, and from 0.331 to 0.406 for Terminal-Bench Hard. Thus, later rounds become lexically more different from the evaluation sets even as median instruction length grows from 87.5 to 121.5 tokens.

B.2 Detailed Synthesis Pipeline and Structural Growth

Figure 6 expands the per-round synthesis pipeline summarized in the main method section. Figure 7 and Figure 8 provide additional structural-growth diagnostics beyond the quantile trends shown in the main text.

Structural metric definitions. All structural measurements are computed statically from each accepted task bundle. Solution length is the number of physical lines in `solution/solve.sh`. Command count approximates shell-command segments on non-empty, non-comment lines after splitting compound lines at pipes, semicolons, `&&`, and `||`, excluding segments headed by shell control keywords. Unique CLI tools are distinct leading command names after skipping environment assignments and wrappers such as `sudo`, `command`, and `env`. Control-flow count records occurrences of `if`, `for`, `while`, `case`, and `function`; file-operation count records command segments led by recognized filesystem-manipulation tools. Verifier assertions are occurrences of the `assert` keyword in `tests/test_state.py`.

Instruction–verifier diagnostics. Figure 9 shows that instructions with at least 80 words increase from 58% in R_1 to 92% in R_{15}, while private-test hints remain absent. Figure 10 provides a complementary lexical measure: median requirement coverage increases from 0.42 to 0.57, low coverage falls from 32.8% to 1.2%, and high coverage rises from 14.3% to 38.0%. Publicly stated requirements therefore become more closely connected to executable verifier checks under this lexical metric. These diagnostics remain proxies and do not establish semantic equivalence.

B.3 Fixed-Solver Outcomes and Diversity Diagnostics

The fixed-solver results show that later tasks are not merely missing one final step. Among failed attempts, the share receiving more than 0.75 partial credit falls from 86.4% in R_1 to 1.2% in R_{15}. At the same time, the share of tasks with average partial credit below 0.50 rises from 0% to 97.5%. Together with the pass@4 decline from 90% to 2.5%, Figures 11 and 12 show that later tasks leave substantially more of the required workflow unresolved rather than merely failing at the final step.

Table 3: Mapping from the high-level domain labels in Figure 14 to the original categories for the 639 oracle-validated bootstrap tasks. Domains are post-hoc merges of overlapping raw categories; they are not native dataset fields.

Domain	#Tasks	Original TerminalWorld categories
Build & Compile	149	`build-system`, `software-development`, `software-engineering`
Scripting & Automation	103	`scripting`, `scripting-automation`
System Admin & Services	87	`process-management`, `system-administration`
Security / Reverse / Crypto	52	`security`
Package & Environment	47	`environment-setup`, `package-management`
Version Control & Release	34	`version-control`
Containers & Kubernetes	34	`container-cloud-native`, `container-operations`, `container-orchestration`
Database & Storage	21	`data-management`, `database`, `database-operations`
Files, Documents & Media	21	`document-processing`, `documentation`, `file-management`, `file-operations`, `text-processing`
Network / Web / API	20	`blockchain`, `networking`
Debug, Test & Quality	12	`debugging`, `debugging-troubleshooting`, `performance-optimization`, `testing`
Cloud, Deploy & DevOps	12	`cloud-infrastructure`, `cloud-services`, `deployment`, `devops`, `distributed-computing`, `monitoring-observability`
Scientific / HPC / Bio	11	`bioinformatics`, `scientific-computing`
Other / Unknown	10	`unknown`
Education / Games / Misc	8	`coursework-exercises`, `games-entertainment`, `program-execution`
Data Processing & Analysis	8	`data-analysis`, `data-processing`
Language Tooling	7	`developer-tools`, `development-tools`, `programming`, `system-programming`, `systems-programming`
ML / AI	2	`ml-training`
Virtualization	1	`virtualization`

Table 4: Task-description distance and direct text-overlap analysis across recursive synthesis rounds. Unigram JSD measures the lexical-distribution difference between each round and the corresponding benchmark, while Max J_5 reports the largest pairwise 5-gram Jaccard similarity. Exact 13-token overlap reports the number of benchmark tasks containing at least one matching normalized 13-token window.

		Unigram JSD			
Round	Median tokens	TB2	LHTB	TB Hard	Max J_5
R_1	87.5	0.358	0.441	0.331	0.0081
R_5	104.0	0.387	0.461	0.359	0.0042
R_{10}	116.0	0.426	0.484	0.394	0.0028
R_{15}	121.5	0.433	0.485	0.406	0.0051
Exact 13-token overlap:		TB2 $0/89$	LHTB $0/46$	TB Hard $0/100$	

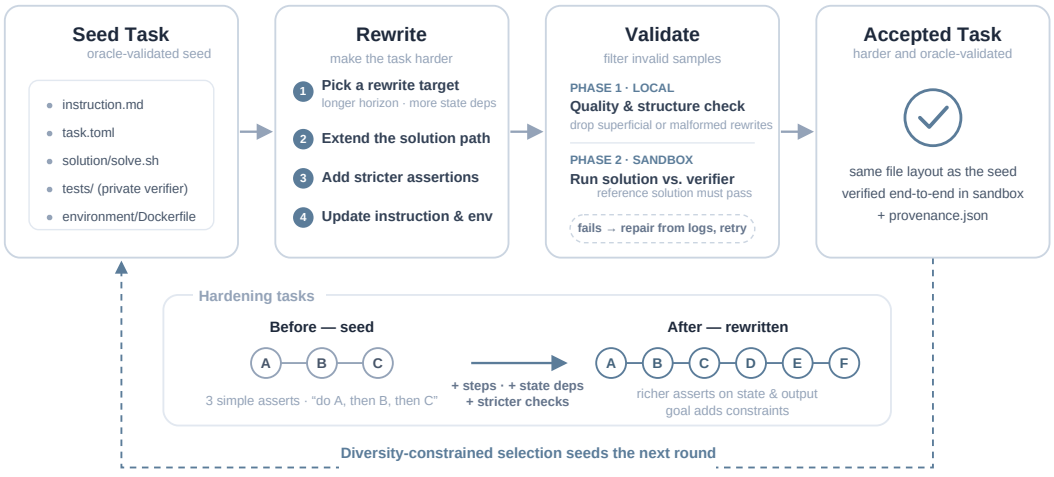

Figure 6: Detailed pipeline of one recursive synthesis round. An oracle-validated seed task undergoes target selection, staged rewriting, local and sandbox validation, and diversity-constrained reseeding.

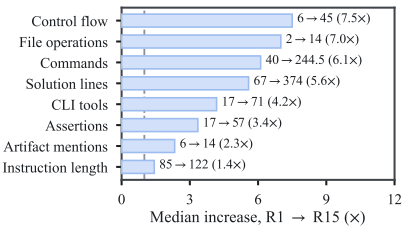

Figure 7: Median expansion factors from R_1 to R_{15} across task-structure metrics. Metrics associated with executable work increase substantially faster than instruction length.

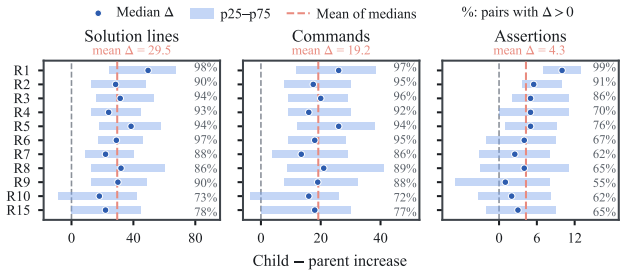

Figure 8: Parent-child changes in solution length, command count, and verifier assertions from R_1 to R_{15}. Most accepted children exhibit positive changes relative to their immediate parents, indicating continued structural growth at the individual task level.

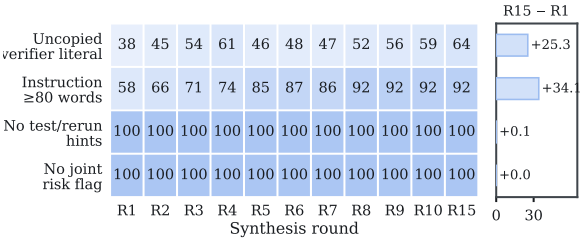

Figure 9: Static lexical audit of public instructions from R_1 to R_{15} on run/loop-stratified samples of 1,000 accepted tasks (seed 42). "Uncopied verifier literal" denotes tasks with at least one eligible verifier literal absent verbatim from the instruction; "instruction ≥80 words" applies a heuristic threshold for flagging very short instructions. The remaining rows exclude private-test/rerun hints and the joint hint/literal-risk condition. Appendix E.1 gives the exact lexical rules and threshold rationale.

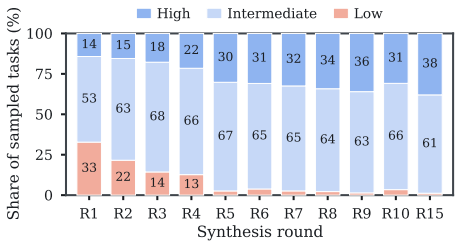

Figure 10: Lexical requirement-to-verifier coverage from R_1 to R_{15} on deterministic 500-task subsamples. Coverage is the fraction of distinct content tokens from requirement-bearing instruction lines that also occur in the verifier. The heuristic low (< 0.35), intermediate ($[0.35, 0.60)$), and high (≥ 0.60) bands are descriptive rather than semantic-validity thresholds; 491–500 tasks per round are usable.

Under diversity-constrained reseeding, this increase in difficulty does not coincide with collapse into one task type. Rewrite-family entropy remains close to its maximum, ancestral-domain balance is stable, and no family exceeds 29% of any round. Although lineage coverage decreases to 60.1% by R_{15}, no lineage contributes more than 0.77% of the tasks, and R_{15} still uses 36 of the 40 rewrite operators. Parent-child distance decreases, median within-round similarity rises from 0.223 to 0.464, and the 95th percentile reaches 0.703. Broad coverage therefore persists under the selection mechanism, although a high-similarity tail emerges in later rounds.

B.4 Training Results and Dataset Comparisons

Figure 19 and Table 5 show that every benchmark mean improves from Base to R_3 for both model sizes. The final gains range from 4.34 to 10.00 points. The largest improvement appears on Terminal-Bench Hard for Qwen3.5-122B-A10B, which rises from 20.00% to 30.00%. This consistency across model sizes and benchmarks suggests that the benefit is not limited to one checkpoint or evaluation setting. Table 6 provides a stronger data-quality control. Under exactly the same token budget, the RST-trained model exceeds the strongest model trained on the five public datasets by 4.12 points on Terminal-Bench 2, 3.34 points on Terminal-Bench Hard, and 1.90 points on Long-Horizon Terminal Bench. Thus, the gains cannot be explained only by using more training tokens, although this comparison does not isolate recursion from other differences in the data.

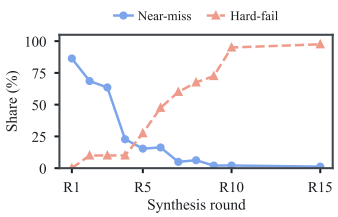

Figure 11: Distributional shift in partial-credit outcomes under two explicitly different denominators. Among failed attempts, the share above 0.75 partial credit falls from 86.4% to 1.2%; among the 200 sampled tasks per round, the share whose four-attempt mean is below 0.50 rises from 0% to 97.5%.

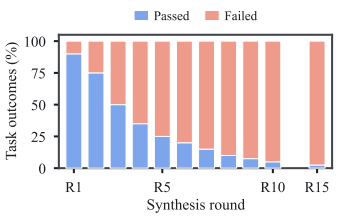

Figure 12: Task-level pass@4 outcomes for DeepSeek-V4-Pro on 200 sampled tasks per round. The pass rate decreases from 90% in R_1 to 20% in R_6 and 2.5% in R_{15}.

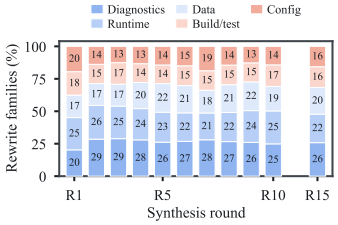

Figure 13: Distribution of accepted tasks across the five rewrite families. Entropy stays close to its maximum, and no family exceeds 29% in any round.

(a) Domain composition (% of tasks)

	Seed	R1	R5	R10	R15
Build & Compile	23.3	22.6	16.4	16.6	17.0
Scripting & Automation	16.1	16.5	15.8	17.8	20.7
System Admin & Services	13.6	14.3	14.8	13.5	11.2
Security / Reverse / Crypto	8.1	6.7	6.0	5.1	6.9
Package & Environment	7.4	8.2	10.7	10.2	8.8
Containers & Kubernetes	5.3	5.1	5.4	5.7	5.7
Version Control & Release	5.3	6.2	8.7	8.2	8.0
Database & Storage	3.3	3.6	4.5	4.1	4.1
Files, Documents & Media	3.3	3.9	3.5	3.9	5.3
Network / Web / API	3.1	3.0	2.2	2.4	0.9
Cloud, Deploy & DevOps	1.9	2.1	2.8	2.6	2.1
Debug, Test & Quality	1.9	1.7	2.2	2.7	2.5
Scientific / HPC / Bio	1.7	1.1	1.6	1.7	1.3
Other / Unknown	1.6	1.5	0.3	0.1	–
Data Processing & Analysis	1.3	1.1	1.2	2.4	2.2
Education / Games / Misc	1.3	1.2	2.3	2.0	2.8
Language Tooling	1.1	0.9	1.1	1.0	0.5
ML / AI	0.3	0.2	–	–	–
Virtualization	0.2	0.1	0.4	0.1	–

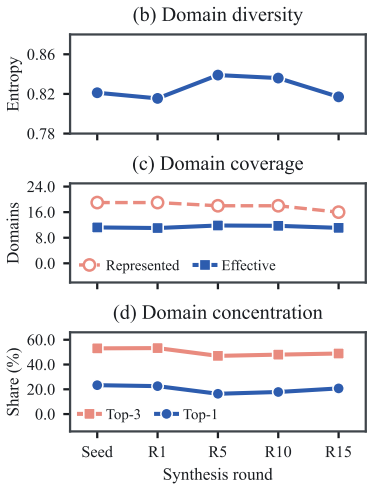

Figure 14: Ancestral-domain composition and stability from the 639 bootstrap seeds through recursive synthesis. (a) Inherited domain proportions for the seed pool and tasks that pass oracle validation at R_1, R_5, R_{10}, and R_{15}. (b) Distribution evenness measured by normalized Shannon entropy. (c) Breadth measured by the number of represented domains and the entropy-derived effective number of domains. (d) Concentration measured by the proportions of tasks belonging to the most frequent domain and the three most frequent domains. The stable entropy and effective count indicate that ancestral-domain balance is preserved through R_{15}.

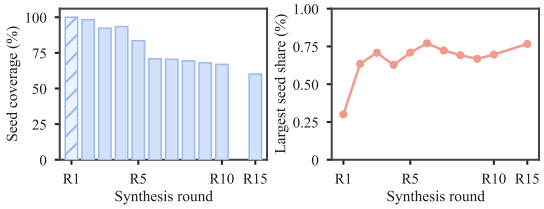

Figure 15: Coverage of bootstrap seeds across recursive rounds. R_{15} retains 218 of the 363 seeds represented in R_1, and no single seed contributes more than 0.77% of the R_{15} tasks.

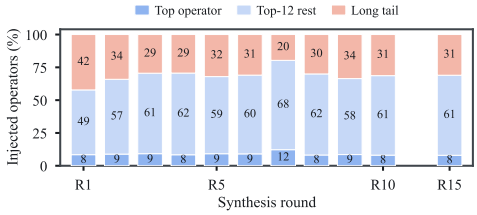

Figure 16: Rewrite-operator coverage from R_1 to R_{15}. Each round uses 31 to 38 of the 40 available operators, and the most frequent operator accounts for at most 12.2% of the tasks.

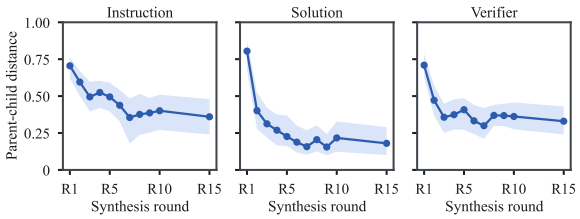

Figure 17: Token-level parent-child novelty in the instruction, solution, and verifier from R_1 to R_{15}. Later-round rewrites retain more content from their parents while continuing to modify all three task components.

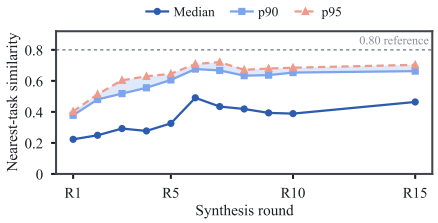

Figure 18: Within-round nearest-neighbor similarity from R_1 to R_{15}. The R_{15} median remains below 0.5, while p95 reaches 0.703, indicating broad task variation with a limited high-similarity tail.

Figure 20 shows that PPO increases mean verifier reward from 0.11 to above 0.14 while increasing average trajectory length from about 20 to more than 30 turns. Higher reward accompanies longer interactions in this run. Table 7 reports downstream gains of 8.61, 9.33, and 3.97 points over the base model. PPO gives the strongest 27B results on Terminal-Bench 2 and Terminal-Bench Hard, but not on the longest-horizon benchmark. Finally, Table 8 shows that RST uniquely combines grounded executable tasks, large-scale trajectory collection, PPO training, and recursive reseeding among the compared systems.

Table 5: Evaluation of Qwen3.5-27B and Qwen3.5-122B-A10B across sequential training stages. After trajectories associated with duplicate task records are removed, each stage retains 10,000 successful trajectories generated by the model that is subsequently fine-tuned. For each model and training-and-evaluation seed (42, 43, or 44), Round 1 starts from Base using the R_1 set; Rounds 2 and 3 continue from the preceding checkpoint using separate sets from R_2 and R_3. Each entry reports the mean $\pm$ sample standard deviation over the three independent runs; Terminal-Bench 2 (TB2) and Terminal-Bench Hard use mean benchmark score, while Long-Horizon Terminal Bench (LHTB) uses mean partial credit.

Model (Non-thinking Mode)	Training stage	Benchmark score (%, mean $\pm$ std.)		
		Terminal-Bench 2	Terminal-Bench Hard	LHTB
Qwen3.5-27B	Base	41.20 ± 1.72	22.67 ± 2.52	18.10 ± 0.89
	Round 1	42.32 ± 4.68	23.00 ± 1.73	21.32 ± 0.86
	Round 2	44.57 ± 4.25	27.33 ± 1.53	21.99 ± 1.39
	Round 3	47.94 ± 2.34	28.33 ± 0.58	22.44 ± 0.40
Qwen3.5-122B-A10B	Base	43.82 ± 2.97	20.00 ± 1.00	18.85 ± 1.16
	Round 1	44.19 ± 1.30	21.33 ± 3.21	20.06 ± 0.32
	Round 2	47.94 ± 1.72	29.33 ± 2.08	20.24 ± 0.60
	Round 3	49.44 ± 1.12	30.00 ± 1.73	23.63 ± 2.76

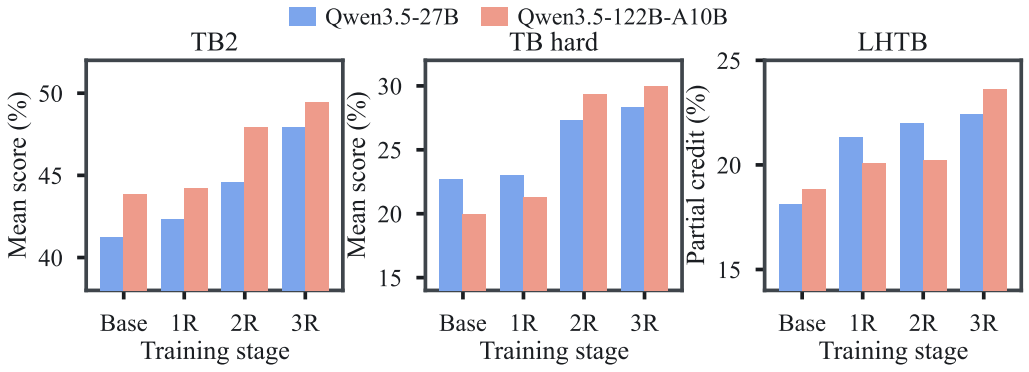

Figure 19: Benchmark performance across sequential supervised fine-tuning stages. After trajectories associated with duplicate task records are removed, each stage retains 10,000 successful trajectories generated by the model that is subsequently fine-tuned. For each model, Round 1 starts from Base using the R_1 set; Rounds 2 and 3 continue from the preceding checkpoint using separate sets from R_2 and R_3. The panels report means over independent training-and-evaluation seeds 42, 43, and 44: benchmark scores on Terminal-Bench 2 and Terminal-Bench Hard and partial credit on Long-Horizon Terminal Bench.

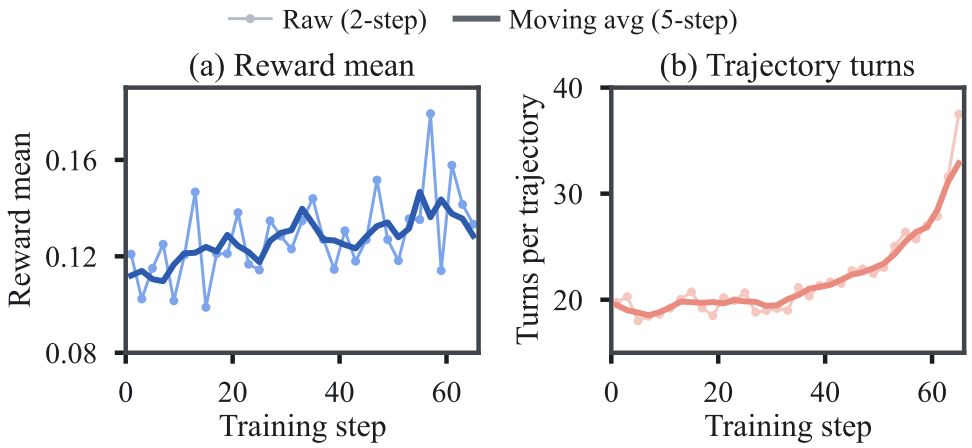

Figure 20: Reinforcement-learning dynamics on the synthesized terminal tasks: (a) mean verifier reward, rising from 0.11 to above 0.14, and (b) mean interaction turns per trajectory, rising from 19–20 to more than 30 turns.

Table 6: Comparison of six terminal-agent SFT datasets under a matched supervision budget of 36,363,275 assistant tokens included in the training loss, corresponding to the full budget of 11,531 unique RST-only trajectories randomly sampled from the pooled R_1–R_3 data. For each dataset we report the number of unique trajectories in the pool used, the effective number of trajectories fed to training (including repetitions for pools smaller than the target budget), mean benchmark scores on Terminal-Bench 2 and Terminal-Bench Hard, and mean partial credit on Long-Horizon Terminal Bench. SFT rows report percent mean $\pm$ sample standard deviation over independent training-and-evaluation seeds 42, 43, and 44. Dataset construction uses fixed random seed 42, and tokens beyond the budget within the last selected trajectory are set to `loss_mask`$= 0$ so all six SFT runs terminate at exactly the same supervision budget. The Qwen3.5-27B Base row reports three evaluations with the same seeds but no post-training.

Training data	Unique trajectories	Trajectory exposures	Assistant tokens in loss	TB2 (Mean $\pm$ Std.)	TB-Hard (Mean $\pm$ Std.)	LHTB (Mean $\pm$ Std.)
Qwen3.5-27B Base (no SFT)	—	—	—	41.20 ± 1.72	22.67 ± 2.52	18.10 ± 0.89
OpenThoughts-Agent-v1-SFT (Raoof et al., 2026)	15,209	15,613	36,363,275	41.95 ± 1.30	23.67 ± 0.58	19.30 ± 1.05
TMax-SFT (Ivison et al., 2026)	10,726	8,314	36,363,275	41.57 ± 1.12	23.67 ± 1.15	19.10 ± 0.75
LiteCoder-Terminal-SFT (Peng et al., 2026)	11,255	3,071	36,363,275	42.70 ± 1.12	24.33 ± 2.08	20.00 ± 1.05
Nemotron-Terminal-Corpus (skill-based) (Pi et al., 2026)	139,841	6,272	36,363,275	39.70 ± 0.65	21.67 ± 0.58	18.30 ± 0.72
SETA-SFT (Shen et al., 2026)	1,112	6,991	36,363,275	40.45 ± 1.12	22.67 ± 1.15	18.80 ± 0.85
RST-only (ours)	**11,531**	**11,531**	**36,363,275**	$\mathbf{46.82 \pm 1.30}$	$\mathbf{27.67 \pm 1.53}$	$\mathbf{21.90 \pm 1.00}$

Table 7: Benchmark performance. Qwen base rows report mean $\pm$ standard deviation over three evaluation runs of each fixed checkpoint. The Qwen3.5-27B-RL row reports three independent PPO training-and-evaluation runs with seeds 42, 43, and 44. Terminal-Bench 2 (TB2) and Terminal-Bench Hard use mean benchmark scores; Long-Horizon Terminal Bench (LHTB) uses mean partial credit. Relative gain compares the Qwen3.5-27B-RL and Qwen3.5-27B Base means.

Model (Non-thinking Mode)	TB2 score (%)	TB-Hard score (%)	LHTB partial credit (%)
DeepSeek-V4-Pro	51.68	36.00	30.00
Qwen3.5-27B Base	41.20 ± 1.72	22.67 ± 2.52	18.10 ± 0.89
Qwen3.5-122B-A10B Base	43.82 ± 2.97	20.00 ± 1.00	18.85 ± 1.16
Qwen3.5-27B-RL[†]	49.81 ± 1.72	32.00 ± 2.00	22.07 ± 1.20
Relative gain	+20.90%	+41.16%	+21.93%

[†] *Detailed results:* For seeds 42, 43, and 44, respectively, TB2 run scores are 48.31, 49.44, and 51.69 (43, 44, and 46 solved tasks out of 89); TB-Hard scores are 30.00, 32.00, and 34.00 (30, 32, and 34 solved tasks out of 100); LHTB partial-credit scores are 20.90, 22.00, and 23.30.

B.5 Case Study: Recursive Growth of a JSON-Diff Regression Task

To illustrate how a task changes across rounds, we follow one complete lineage with checkpoints available from the bootstrap seed through R_{15}. This example is illustrative rather than statistically representative. The original task asks the agent to run the `gendiff` command-line tool on fixed pairs of JSON fixtures and save the generated reports. Later rounds retain this objective but add configuration files, regression cases, failure diagnosis, release-note updates, and executable tests.

Table 9 shows five checkpoints in this lineage. The added difficulty comes from new executable requirements rather than longer instructions. Later versions introduce a task matrix in `tasks.json`, checks for invalid configuration, a new regression case, and consistency constraints across fixtures, expected counts, reports, and tests. By R_{15}, the agent must repair the configuration and fixture data, regenerate all reports, reconcile observed and expected counts, and pass the project test suite.

This lineage provides a concrete example of the complexity growth shown in the main text and in Figure 8. The core objective remains unchanged, but each round adds more executable work. The agent must inspect workspace evidence, diagnose inconsistent state, repair configuration and data, regenerate artifacts, and pass executable tests. As a result, the reference solution grows from 10 to 347 non-empty shell lines without changing the underlying task domain.

Table 8: Comparison of representative task datasets and environment-generation pipelines. Task and trajectory counts are reported separately because executable environments and agent interaction traces are distinct data units. Prior-work properties follow the corresponding papers and releases.

Dataset	Domain	Tasks†	Trajectories‡	Grounded?	Executable?	Adaptive?	Used for RL?	Reseeding?
WizardLM (Xu et al., 2024)	General	–	250,000[a]	×	×	×	× (SFT)	×
SWE-Gym (Pan et al., 2025)	SWE	2,438	N/R	✓	✓	×	× (SFT)	×
OpenThoughts-Agent-v1-SFT (Raoof et al., 2026)	Terminal	N/R	15,209	△	✓	×	× (SFT)	×
TermiGen (Zhu et al., 2026)	Terminal	3,500+	3,291	×	✓	×	× (SFT)	×
RLVE-Gym (Zeng et al., 2026)	Math/Algo	400	N/R	✓	×	✓	✓ (DAPO)	×
ScaleEnv (Tu et al., 2026)	Tool-Use	2,560	N/R	×	✓	×	✓ (GRPO)	×
Endless Terminals (Gandhi et al., 2026)	Terminal	3,255	N/R	×	✓	×	✓ (PPO)	×
Terminal-Corpus (Pi et al., 2026)	Terminal	N/R	254,000+	✓	✓	×	× (SFT)	×
TMax-15K (Ivison et al., 2026)	Terminal	14,600	N/R[b]	×	✓	×	✓ (GRPO)	×
SETA (Shen et al., 2026)	Terminal	4,567	1,112[c]	✓	✓	✓	✓ (GRPO)	×
RST (ours)	**Terminal**	**37,484**	**327,189**	✓	✓	△	✓ **(PPO)**	✓

†Task counts report reusable task or environment instances rather than model executions. ‡Trajectory counts report collected agent interaction traces. N/R indicates that no fixed trajectory-corpus size is reported. [a]WizardLM reports single-turn instruction–response examples rather than terminal-agent trajectories. [b]TMax separately reports 16.5k SFT trajectories generated from an additional 2.2k-environment warm-start set. These are not counted as trajectories from TMax-15K. [c]SETA retains 1,112 successful SFT trajectories from 1,488 collected rollouts. *Grounded* denotes construction anchored to existing source artifacts or seed tasks; *executable* denotes runnable task environments; *adaptive* denotes generation or selection that responds to model behavior; *used for RL* denotes reported reinforcement-learning use; and *reseeding* denotes reuse of generated tasks as inputs to later synthesis rounds. RST is marked mixed for adaptivity because behavior-derived rollout metrics are optional in seed selection rather than required in every synthesis round. ✓, ×, and △ denote yes, no, and mixed support, respectively.

Table 9: Case study of one exact recursive lineage. The task keeps the same core objective, generating reliable JSON-diff regression outputs, while later rounds add configuration, failure diagnosis, release-note updates, and count reconciliation. The last column reports the number of non-empty lines in the reference `solution/solve.sh`.

Stage	Public task	Added context	Agent work required	Lines
Seed	Generate six JSON-diff reports from fixed fixture pairs.	Manual CLI example with fixed input files.	Run `gendiff` on two JSON fixture pairs in three output formats.	10
R1	Generate all diff outputs specified by `tasks.json`.	The comparison matrix is moved into configuration.	Read `tasks.json`, batch-run all comparisons, and write a validation report.	66
R5	Repair `tasks.json` using `check_config.sh`, then regenerate the diffs.	The configuration may be invalid and must be diagnosed.	Run the checker, interpret failures, fix the task list, rerun the suite, and aggregate change counts.	219
R10	Use `CHANGELOG.md` to add the new nested-fixture regression case.	A release note introduces a new fixture and required comparison.	Update the regression matrix, generate the new output, and satisfy the local report validator.	264
R15	Repair inconsistent configuration and fixture data; reconcile expected diff counts.	Configuration, fixtures, expected counts, and tests must agree.	Fix `tasks.json` and JSON fixtures, regenerate every diff, write diagnosis and validation reports, and pass tests.	347

C Recursive Synthesis Implementation

The recursive synthesis system is manifest-driven, and every transformation, validation result, and round-to-round transition is recorded explicitly. Each seed-manifest entry identifies the task directory and preserves its task ID, source round, source loop, source cohort (the generation-run group in which it was produced), original parent, rewrite family, rewrite operator, and repair history. Before transformation, the system verifies that the seed contains the required task components, including the public instruction, task configuration, reference solution, verifier entry point, verifier checks, and environment definition. The complete seed directory is then copied into an isolated output directory. Generation is restricted to six task files: `instruction.md`, `solution/solve.sh`, `tests/test.sh`, `tests/test_state.py`, `environment/Dockerfile`, and `task.toml`. File writes outside this allowlist or outside the copied task directory are rejected.

Rewrite targets are selected using a taxonomy of 40 operators grouped into five families: environment and runtime substrate, build and execution, data and artifact processing, configuration and state migration, and diagnostics and auditing. For each seed, a local compatibility scan examines bounded representations of its instruction, solution, verifier, configuration, and environment files to determine which transformations are supported. It scores operators according to observable signals such as package manifests, structured-data formats, build scripts, command-line interfaces, archives, databases, logs, permissions, services, and version constraints. Operator selection combines this local compatibility score with a family-balance term and an inverse-frequency penalty for operators already assigned within the current batch. This prevents a small number of broadly applicable operators from dominating the generated pool. The selector records one preferred operator, up to five feasible alternatives, and operators that should not be applied to the seed. An additional model-based ranking step evaluates whether the preferred operator is natural, safe, and sufficiently supported by the seed. It may retain the preferred operator or replace it with one of the recorded alternatives, but it cannot introduce an operator that was excluded by the local scan.

For seed i and retained operator o, the implementation computes

$$w_i(o) = A_i(o)\frac{\max\big(0.25,\ 1 + 0.2N_i - C_{f(o),i}\big)}{1 + C_{o,i}}\kappa_o,$$

where $A_i(o)$ is the local compatibility score, N_i is the larger of one and the number of operators already assigned in the batch, and $C_{f(o),i}$ and $C_{o,i}$ count prior assignments to the same family and operator. The fixed correction is $\kappa_o = 0.35$ for `config_data_consistency` and 1 otherwise. This score is a deterministic balancing heuristic, not a learned objective.

The selected operator is first converted into an explicit transformation contract rather than being applied directly. This contract specifies the original behavior that must be preserved, the new terminal-native requirement, the expected sequence of inspection, derivation, execution, validation, and finalization, and the observable artifacts associated with each stage. It also defines discoverable evidence, intermediate outputs, final outputs, verifier checks, and conditions that reject empty, stale, hard-coded, or verifier-specific shortcuts. The contract requires at least four distinguishable checks covering evidence discovery, intermediate-state validity, final semantic correctness, and shortcut rejection. It separately records what must appear in the public instruction and what may remain implicit because it can be discovered from local documentation, configuration files, fixtures, logs, source files, or normal command output.

Task construction then proceeds through ordered, file-scoped stages. The reference-solution stage extends `solution/solve.sh` so that it completes both the preserved seed behavior and the new requirement from the initial sandbox state. The verifier stage derives checks from the transformation contract rather than copying incidental commands from the solution. It checks semantic content and state, preserves relevant checks from the parent task, and rejects solutions that produce only placeholder artifacts. The instruction stage expresses the user-visible goal, a small number of fair starting points, and the required deliverable without exposing private verifier paths, exact solution commands, or a complete acceptance checklist. An environment-alignment stage is invoked only when the extension requires a local dependency, fixture, configuration, or metadata change. After these stages, a cross-file consistency pass compares the instruction, solution, verifier, environment, and transformation contract. It repairs discrepancies such as artifacts produced by the solution but

omitted from the contract, verifier requirements unsupported by public evidence, or instruction requirements that are not evaluated.

Two deterministic gates are applied before sandbox execution. The generation-quality gate verifies that the candidate represents a substantive cross-file transformation. In the current implementation, at least three task files must change, the reference solution must contain at least eight changed lines, and the verifier must contain at least twelve changed lines. The same gate rejects instructions that expose private test paths, resemble step-by-step command recipes, enumerate excessive implementation details, exceed 180 words, or expand to more than 1.6 times the length of the parent instruction. The static-preflight gate checks Docker build consistency, required task files, critical artifact paths, and the correspondence between artifacts required by the verifier and artifacts produced by the solution or environment. It also verifies that evidence declared discoverable in the contract is present in the task bundle. Failures are recorded separately as contract, generation-quality, or static-preflight failures, which allows these losses to be distinguished from failures during executable validation.

Candidates that pass the deterministic gates are executed with their reference solutions in fresh Daytona sandboxes through Harbor. The private verifier then grades the resulting environment. A candidate passes oracle validation only when the verifier returns a reward of one and the trial contains no execution exception. Other outcomes are classified as zero reward, build failure, timeout, runtime error, excessive output, or unknown failure. Failed candidates from configured repair categories enter a bounded feedback-repair procedure. The repair model receives the transformation contract together with truncated build logs, execution logs, verifier results, and failure metadata. It must identify a specific failure type and may modify only the six allowed task files. Every repaired candidate is subjected to static preflight again and is then revalidated in a new sandbox. The reported configuration permits at most two feedback-repair rounds, preventing repeated unconstrained regeneration from obscuring the original transformation. The final manifest includes only candidates that pass the last validation attempt and records their initial result, final result, rewrite metadata, repair count, task directory, parent identifier, and provenance.

Synthetic task generation uses DeepSeek-V4-Pro through the OpenAI-compatible DataEval API proxy with temperature 0.2, top-p 0.95, a 900 s request timeout, and at most two retries. R_1 starts from the 639 bootstrap tasks; each later round selects up to 1,000 seeds, then applies fresh-sandbox Daytona validation and at most two repair rounds. Earlier runs use generation, Daytona, and repair concurrency of 64, 100, and 32, respectively; later high-throughput runs use 500, 500, and 128. These concurrency settings affect throughput but not task selection or acceptance criteria.

Round-to-round orchestration begins by reading the `passed_tasks` records from every `final_manifest.json` produced by the preceding round. Records are deduplicated by resolved task path while retaining the source run, loop, cohort, family, operator, repair count, and original parent. The resulting `pool_manifest.jsonl` is the complete candidate pool for the next round. Seed selection assigns each candidate a score based on static risk, environment cost, rewrite metadata, and validation history. Long build timeouts and resource-intensive environment features receive penalties. Tasks requiring one repair remain eligible because they have subsequently passed executable validation, while candidates requiring repeated repair receive a stability penalty. When rollout statistics are available, the selector favors tasks with nontrivial but nonzero success rates, low infrastructure-error rates, and intermediate interaction lengths. Tasks with very high pass rates are penalized as potentially too easy, while tasks with almost no successful executions or high error rates are penalized as potentially unstable.

The selector initializes each candidate at 100 and adds the adjustments in Table 10. These values are fixed engineering heuristics rather than learned parameters or theoretically derived weights. Candidates below the minimum score or triggering a hard exclusion are removed before diversity-constrained selection.

After scoring, candidates are sorted deterministically rather than sampled uniformly and are selected subject to diversity constraints over original parent, task category, rewrite family, and source cohort. The standard configuration targets 1,000 seeds and allows at most four tasks from one original parent, 160 tasks from one category, 320 tasks from one rewrite family, and 280 tasks from one source cohort. If these constraints prevent the target size from being reached, the controller relaxes them according to a predetermined schedule and records the cap configuration used for each attempt. It does not silently fill the remaining positions without diversity constraints. Across the recorded

Table 10: Complete seed-priority scoring and exclusion rules. Rows involving rollout statistics are applied only when those statistics are available.

Signal	Condition	Adjustment or action
Static risk	Each unit of aggregate static-risk score	-6
Build timeout	900–1,199 s; at least 1,200 s	-8; -18
Heavy environment	Each flagged heavy environment feature	-8
Rewrite family	Diagnostics; build; configuration; data; environment; unknown	$+14, +10, +9, +6, -4, 0$
Rewrite operator	`git_history_diff_forensics` or `verifier_failure_interpretation`	$+10$
Rewrite operator	`evidence_bundle_generation`	$+9$
Rewrite operator	`process_endpoint_inspection` or `unit_test_failure_repair`	$+8$
Rewrite operator	`artifact_inventory_reconciliation`, `config_data_consistency`, or `manifest_lockfile_reconciliation`	$+7$
Rewrite operator	`schema_content_validation` or `state_migration_transform`; any other operator	$+6$; 0
Repair history	One repair; two repairs	$+4$; -4
Rollout sample count	n matched rollout samples	$+\min(8, n)$
Rollout pass rate	$0.15 \le p \le 0.60$; $p > 0.80$; $p < 0.05$	$+30$; -25; -18
Rollout error rate	$e \le 0.10$; $e \ge 0.25$ with sufficient samples	$+12$; hard exclusion
Median interaction length	$20 \le \ell \le 50$; $\ell < 10$; $\ell > 60$	$+25$; -20; -15
Final threshold	Total score below 20	Hard exclusion

selections, selected scores range from 92.0 to 128.0, with a median of 117.0. The selected task bundles are then materialized into a local seed directory, and a new manifest is written with both the materialized path and the original source path. Materialization verifies that every selected bundle can be copied successfully and that its task configuration is present before synthesis is launched.

The selector and materializer additionally protect recursive ancestry. Parent identifiers are recovered from the explicit `parent_seed`, `original_seed`, `source_task_id`, task ID, and source path fields. Because recursively generated task names may exceed filesystem limits, long names are compacted to at most 96 characters while preserving the original bootstrap-parent identifier and appending a stable hash. For manifests containing at least 20 records, the controller requires at least 80% of records to retain recoverable bootstrap lineage. The same check is applied before selection, after selection, and after materialization. A failed check terminates the round before generation. This safeguard is necessary because loss of parent identity would make the per-parent cap ineffective and would invalidate lineage-coverage and parent-child novelty analyses.

D Operator Taxonomy

The synthesis factory organizes rewrite targets according to five sources of executable difficulty: the runtime environment in which commands operate, the build and execution process, the artifacts produced or transformed, the configuration or persistent state being maintained, and the diagnostic evidence required to identify and resolve failures. We choose these five families because they capture the five distinct locations at which additional difficulty can be introduced into a terminal task. *Environment and runtime substrate* covers the execution conditions that must be established before a task can run, such as dependencies, permissions, paths, processes, and resource constraints. *Build, test, and execution* covers the command sequence and program behavior required to complete the task. *Data, artifact, and report processing* covers transformations whose primary objective is to produce or validate concrete outputs. *Configuration and state migration* covers persistent or cross-file state that must remain consistent across versions, executions, or recovery operations. *Diagnostics, audit, and forensics* covers tasks in which the central challenge is to infer the required action from logs, traces, failures, permissions, or other system evidence.

This division was selected according to three criteria. First, it covers the principal sources of complexity encountered in terminal interaction, from execution setup to evidence-based diagnosis. Second, every family can be instantiated as observable file, process, command, or system-state changes, making generated tasks executable and automatically verifiable. Third, the families describe trans-

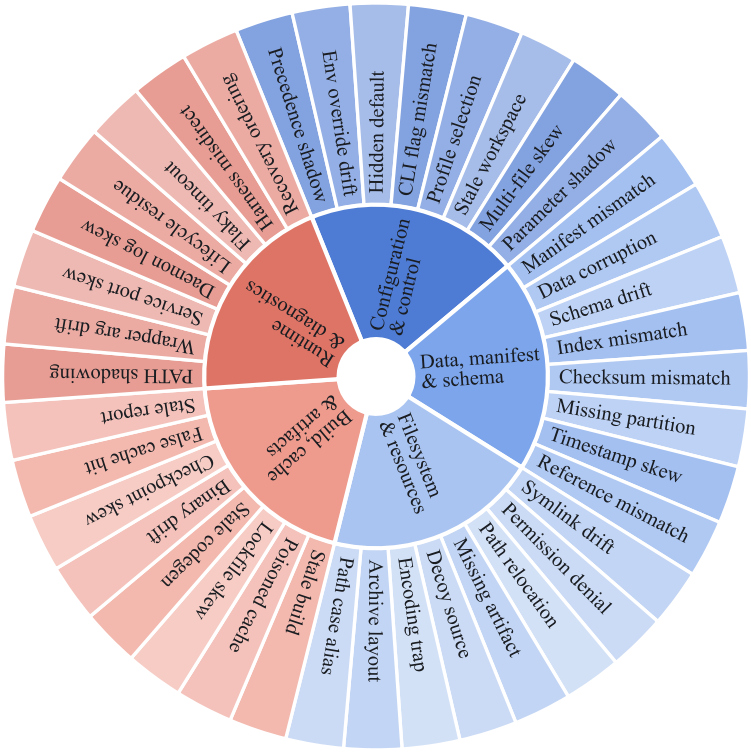

Figure 21: Rewrite-operator taxonomy used for target selection. The operators cover environment, build, data, configuration, and diagnostic workflows; Appendix D provides their full definitions.

formation mechanisms rather than application domains such as software engineering, data processing, or security, whose boundaries overlap and do not specify how a seed should be made harder.

Finer distinctions are represented by eight operators within each family, while assigning each rewrite to its dominant mechanism keeps the round-level distribution measurable and prevents an excessively fragmented taxonomy. Each operator is therefore accompanied by an executable card specifying compatible seed affordances (observable files, tools, or runtime evidence that support the operator), construction patterns, evidence sources, expected artifacts, verifier and instruction strategies, and shortcut-rejection criteria. Figure 21 summarizes the five families, and Table 11 lists all operators used by the local affordance scan and contract-construction stage.

E Local Filters, Preflight, and Repair Policy

Generation quality filter. Before sandbox execution, the pipeline checks whether a rewrite is substantive and passes automated public-contract checks. A candidate is rejected if it uses an unknown operator, changes fewer than three tracked files, modifies the solution by fewer than eight lines, or modifies the verifier by fewer than twelve lines. These thresholds prevent superficial or verifier-only rewrites and are not used as difficulty measures. The filter also rejects instructions that expose private test paths, prescribe test-driven acceptance loops, exceed 180 words or $1.6\times$ the seed length, enumerate excessive paths or commands, or reveal verifier details. Borderline cases are retained with warnings; passing this filter does not guarantee human-judged public solvability or release readiness.

Static preflight. Static preflight checks cross-file executability before costly sandbox validation. It detects malformed Dockerfiles, missing `COPY` sources, and other environment inconsistencies. It also verifies that artifacts required by the contract or verifier are produced by the solution or provided by the environment, and that promised evidence is discoverable from the task files. Missing

Table 11: Rewrite operator taxonomy used by RST. Family IDs match the implementation; operator IDs are the soft targets assigned before staged rewrite.

Family	Operator ID	Definition
environment runtime substrate	dependency_version_alignment	Align dependency/runtime versions and verify the toolchain.
	path_workdir_alignment	Resolve working-directory and path/entrypoint mismatches.
	permission_executable_alignment	Repair executable bits, ownership, or access needed by the workflow.
	environment_variable_resolution	Infer/configure env vars, locale, timezone, shell init, or profiles.
	toolchain_availability_check	Ensure required compilers/interpreters/CLI tools exist.
	container_build_alignment	Align Dockerfile, build context, and image setup.
	service_process_lifecycle	Start/stop/inspect local services, daemons, or ports.
	resource_cleanup_and_limits	Handle temp files, timeouts, log growth, and cleanup.
build/test execution workflow	compile_link_package_workflow	Compile, link, package, or assemble a runnable artifact.
	unit_test_failure_repair	Run unit tests, interpret failures, and complete the workflow.
	integration_test_workflow	Coordinate multiple components and validate integrated behavior.
	cli_argument_behavior	Validate CLI flags, stdio, exit codes, and argument behavior.
	lint_format_static_check	Run lint/format/type-check and resolve surfaced issues.
	runtime_error_debug_loop	Diagnose runtime exceptions/stack traces and repair behavior.
	build_artifact_generation	Produce a binary/package/bundle and verify usability.
	performance_or_benchmark_smoke	Run a bounded benchmark/smoke check and record a result.
data/artifact report processing	format_conversion_validation	Convert formats while validating semantic preservation.
	schema_content_validation	Validate schema, fields, types, ranges, or completeness.
	aggregation_summary_report	Compute aggregates and produce a summary/report.
	artifact_inventory_reconciliation	Reconcile outputs, manifests, and directory inventories.
	archive_compression_extraction	Extract/pack/inspect archives such as tar/zip/gzip.
	dedup_sort_normalization	Deduplicate, sort, normalize, or canonicalize artifacts.
	checksum_hash_provenance	Generate or verify hashes/checksums/provenance.
	media_or_binary_metadata_processing	Inspect lightweight media/binary metadata.
configuration state migration	config_data_consistency	Reconcile configuration and data/source consistency.
	manifest_lockfile_reconciliation	Align manifests, lockfiles, or package metadata.
	state_migration_transform	Migrate state across formats/versions/layouts.
	cache_index_regeneration	Regenerate caches, indexes, or derived metadata.
	database_or_file_state_initialization	Initialize and validate file-backed/DB state.
	template_render_consistency	Keep templates and rendered outputs consistent.
	profile_feature_flag_selection	Select/validate profiles or feature-flag branches.
	backup_rollback_idempotency	Support backup, rollback, and idempotent reruns.
diagnostics audit forensics	log_error_diagnosis	Inspect logs/stderr, identify cause, complete workflow.
	trace_event_correlation	Correlate logs/events/timestamps across sources.
	verifier_failure_interpretation	Use local validation output to infer missing state.
	security_permission_audit	Audit permissions, sensitive files, or unsafe config.
	data_quality_anomaly_investigation	Investigate missing/duplicate/anomalous data.
	process_endpoint_inspection	Inspect local processes, ports, sockets, or endpoints.
	git_history_diff_forensics	Use git history/diff to diagnose or produce state.
	evidence_bundle_generation	Create a verifiable evidence bundle with logs/reports.

shell safety conventions, such as a shebang or `set -e`, produce warnings rather than rejection. Candidates with hard preflight failures are not submitted to Daytona.

Sandbox oracle validation and feedback repair. Candidates passing both static gates are executed by the reference solution in a fresh Daytona sandbox and accepted only if the verifier returns full reward without an execution exception. Failures are classified as zero reward, build failure, runtime error, timeout, or unknown failure. Recoverable failures may enter a bounded repair loop conditioned on the task contract and Daytona logs. Repair may modify only the solution, verifier, instruction, environment, and task configuration files. It may correct implementation or contract inconsistencies, but cannot weaken semantic checks, remove shortcut protection, or expose private requirements in the instruction. Each repaired task must pass static preflight and Daytona validation again. Candidates that still fail after the configured repair rounds are excluded from the accepted manifest.

E.1 Task Quality Control and Validation

Requirement discoverability. The pipeline identifies every condition that can change the verifier reward and checks whether the agent can know that condition before acting. Each condition is handled in one of four ways:

- A semantic requirement with no public source must be stated directly in the instruction.
- A requirement documented in a local README, specification, configuration, or fixture may remain there, but the instruction must provide a clear starting point for finding it.
- A secondary deliverable must be named in the instruction, while its detailed schema may remain in discoverable workspace documentation.
- A check that only rejects placeholders, stale outputs, or hard-coded answers need not be stated because it adds no new semantic requirement.

The pipeline is designed to reject verifier requirements that are neither stated nor discoverable.

Public-instruction boundary. The public instruction specifies the task objective and every reward-relevant condition that cannot be discovered from the workspace. It does not expose verifier implementation details, private test paths, or the commands used by the reference solution. When an exact value, path, or schema is already available in a local README, specification, configuration, or fixture, the instruction identifies that evidence source instead of repeating its contents. This keeps the task concise without hiding information required for success.

Static instruction-audit indicators. The public-instruction audit in Figure 9 uses a run/loop-stratified sample of 1,000 accepted tasks per round, drawn with random seed 42. The four indicators are lexical heuristics rather than semantic validity tests. *Uncopied verifier literal* is the share of tasks containing at least one eligible verifier literal that is absent verbatim from the public instruction. Eligible literals are 12 or more characters long, contain at least three alphabetic characters, do not begin with a path separator, and are extracted from string equality or membership assertions and named expected-value constants in `tests/test_state.py`. The length and alphabetic-content filters remove short or generic constants that provide little evidence of direct textual exposure. *Instruction $\geq$80 words* is the share of instructions meeting an operational threshold used to flag unusually terse task specifications; 80 words is a heuristic cutoff rather than a general quality boundary. *No test/rerun hints* is the share that does not direct the agent to inspect private test paths or files, inspect test failures, or rerun until tests pass. *No joint hint/literal-risk flag* is the share that does not simultaneously contain such a test/rerun hint and an eligible verifier literal absent from the instruction. These indicators measure direct textual exposure and instruction length; the separate discoverability checks above enforce semantic contract validity.

Lexical requirement-to-verifier coverage. Figure 10 uses a deterministic, evenly spaced subsample of 500 tasks from each round's cached audit sample. Tasks without both `instruction.md` and `tests/test_state.py`, or without any extracted requirement token, are excluded, leaving 491–500 usable tasks per round. A line is treated as requirement-bearing if it begins with a bullet

or numbered-list marker or contains an obligation or action term such as *must*, *should*, *required*, *ensure*, *create*, *write*, *update*, or *report*. Let Q_t be the set of distinct lowercased content tokens from these lines after removing short tokens, numbers, and stopwords, and let V_t be the corresponding token set from the verifier. The lexical coverage score is

$$C(t) = \frac{|Q_t \cap V_t|}{|Q_t|}.$$

For descriptive comparison across rounds, we use fixed low-coverage ($C(t) < 0.35$), intermediate-coverage ($0.35 \leq C(t) < 0.60$), and high-coverage ($C(t) \geq 0.60$) bands. These heuristic cutoffs provide coarse summaries of sparse, partial, and majority lexical overlap; they are not calibrated semantic-validity thresholds. The score is a reproducible lexical proxy for whether publicly stated requirements are reflected in executable checks, but it does not establish semantic equivalence between an instruction requirement and a verifier condition.

Contract-granularity variants. To study the effect of instruction specificity, an optional analysis stage creates five instruction-verifier variants while keeping the environment and reference solution fixed. The variants range from a goal-level exploratory contract to a nearly complete statement of the acceptance conditions. At lower granularities, a condition omitted from the instruction must also be removed from the verifier. At higher granularities, retained verifier conditions must be stated explicitly or linked to discoverable workspace evidence. Consequently, no variant evaluates an undisclosed semantic requirement.

F Synthesis Prompt Templates

This appendix presents the core prompt templates used by RST. All fixed instructions are reproduced verbatim from the implementation. Run-dependent inputs, including seed-task files, operator cards, task contracts, and Daytona feedback, are replaced with angle-bracket placeholders labeled by the functions that supply them at execution time.

SYSTEM Prompt

```
You generate high-quality TerminalWorld task transformations.

Return ONLY valid JSON. Do not use markdown.

Rules:
- Keep the original task recognizable.
- Rewrite only files explicitly requested in this step.
- Return full file contents, not patches.
- Do not add benchmark canary text.
- Do not add network-only requirements or heavyweight new dependencies unless the seed
    already naturally uses them.
- The final task must be solvable by the rewritten solution and verifiable by tests.
- The user instruction must not leak the full solution or private test harness paths.
- The rewritten task should preserve the seed's main workflow and add one realistic
    terminal-native subgoal.
- Prefer tasks that require filesystem inspection, CLI composition, config/data
    validation, log/error interpretation, and observable artifact checks.
- Do not turn the task into a narrow "fix this bug" prompt; any repair/debugging should
    be one part of a larger goal-oriented workflow.
- Keep instruction.md a compact public goal, not an acceptance rubric. Only publish
    undiscoverable hard requirements; put discoverable details in workspace docs and
    point to them. Never leave acceptance criteria only inside tests/, and never dump
    path inventories, field schemas, or step checklists that invite reward hacking.
```

AFFORDANCE RANKING Prompt

```
AFFORDANCE RANKING.

Role:
Evaluate whether the proposed rewrite target is natural for this seed task. This is a
    ranking step only; do not rewrite files.
```

```
Inputs:
- A preferred operator selected by the pipeline using local file signals and diversity
    quotas.
- Several fallback operators from the same local feasibility scan.
- The current seed task files.

Decision rules:
- Mark the preferred operator high only if it can be supported by existing seed
    affordances or small local evidence additions.
- If the preferred operator would be artificial, rank fallback operators that better fit
     the seed.
- Do not invent a new taxonomy outside the provided operators.
- Prefer operators that require terminal exploration and verifiable artifacts, not
    superficial wording changes.
- Judge feasibility against the operator card's seed_affordances, construction_pattern,
    expected_artifacts, and anti_shortcut_strategy.

Return schema:
{
  "status": "ok|blocked",
  "ranked_operators": [
    {"operator": "operator_id", "fit": "high|medium|low", "reason": "short reason"}
  ],
  "blocked_operators": [
    {"operator": "operator_id", "reason": "why it is artificial or unsafe for this seed
     "}
  ],
  "rationale": "short summary"
}

Rewrite target context:
<json.dumps(target_context, ensure_ascii=False, indent=2)>

Seed task:
<json_task_context(task_dir)>
```

STEP 0: Task Transformation Contract Prompt (Part 1/3)

```
STEP 0: task transformation contract. Do not rewrite files in this step.

Role:
Design one controlled, verifiable TerminalWorld rewrite contract. The task must preserve
     the seed's core workflow and add one terminal-native requirement that increases
    workflow depth.

Rewrite target protocol:
- Use the preferred operator if it is natural for this seed.
- If the preferred operator is artificial or unsafe, choose one fallback operator and
    explain why.
- Do not choose blocked operators.
- Do not collapse to config/data consistency unless the seed genuinely exposes config/
    data relationships.
- The operator should determine the main transformation mechanism, but the task must
    still feel like a normal terminal workflow.

Operator card protocol:
- Treat the selected operator card as a simple construction recipe, not a label.
- Use the card's construction_pattern to build a concrete task_chain.
- Use the card's evidence_sources to decide where hidden details should be discoverable.
- Use the card's expected_artifacts to define evidence, intermediate, and final
    artifacts.
- Use the card's verifier_strategy and anti_shortcut_strategy to define reward_checks.
- Use the card's instruction_strategy as a style hint for tone/compactness, not as a
    license to hide acceptance criteria.

Quality criteria:
- The task starts from the original seed initial state, not from a halfway failure state
    .
- The new requirement is user-visible through artifacts, command output, state, or
    reports.
- Details may be hidden from instruction only when they remain discoverable through
    local files, scripts, configs, fixtures, logs, or normal command output that the
    instruction fairly points to.
- The required work should involve inspection, derivation, execution, validation, and
    finalization, not a single deterministic file write.
```

```
- The verifier can check semantic correctness and reject shortcuts, but must not invent
    silent acceptance criteria.
- The contract must define a short task_chain and at least four observable reward_checks
     for dense RL-style feedback.
- The public instruction must remain solvable without verifier access, but stay compact
    to avoid reward hacking: mention the goal, fair start, main deliverable/completion
    signal, and only those exact constants that are not discoverable from workspace
    docs.
- Prefer instruction_boundary.hide + workspace evidence over inlining schemas, path
    inventories, or operational steps. Never hide a semantic check that exists only in
    the verifier with no discoverable evidence.

Hard constraints:
- superficial refactors of solution style;
- merely strengthening tests without changing the user-visible task;
- hidden-only requirements that instruction never implies and the agent cannot discover
    from workspace evidence;
- adding large dependencies, internet access, services, or long-running workloads;
- creating a task that is just "fix a bug";
- telling the agent to inspect or rerun /tests, tests/test.sh, or pytest as the
    acceptance loop.
```

STEP 0: Task Transformation Contract Prompt (Part 2/3)

```
Return schema:
{
  "status": "ok|blocked",
  "rewrite_family": "selected family id",
  "rewrite_operator": "selected operator id",
  "operator_fit": "preferred|fallback",
  "why_fit": "short reason why this operator naturally fits the seed",
  "goal": "one concise user-facing goal for the rewritten task",
  "preserved_workflow": "what original workflow remains",
  "new_requirement": "the added terminal-native requirement",
  "task_chain": [
    {"stage": "inspect", "artifact_or_state": "local config/schema/log/source/build
     output to inspect"},
    {"stage": "derive", "artifact_or_state": "rule/schema/path/expected state to infer
     from local evidence"},
    {"stage": "execute", "artifact_or_state": "intermediate or final artifact/state to
     create by running or editing"},
    {"stage": "validate", "artifact_or_state": "local validation/check result or
     parseable evidence"},
    {"stage": "finalize", "artifact_or_state": "final deliverable or checked state"}
  ],
  "expected_artifacts": [
    {"path": "relative or absolute path used by the task", "role": "evidence|
     intermediate|final", "requirement": "observable content/state requirement"}
  ],
  "reward_checks": [
    {"name": "check_01_required_evidence", "checks": "discoverable evidence source
     exists and is used or referenced"},
    {"name": "check_02_intermediate_artifact", "checks": "intermediate artifact/state is
      present, parseable, and non-empty"},
    {"name": "check_03_final_semantics", "checks": "final output/state satisfies the
     semantic task goal"},
    {"name": "check_04_no_shortcut", "checks": "reject placeholder, empty, stale,
     hardcoded, or verifier-only outputs"}
  ],
```

STEP 0: Task Transformation Contract Prompt (Part 3/3)

```
  "instruction_boundary": {
    "mention": ["public goal", "at most 1-2 fair starting points", "main final
     deliverable or completion signal", "only undiscoverable hard acceptance constants
     "],
    "hide": ["step-by-step commands", "path/field inventories", "schema/format dumps", "
     oracle repair recipe", "anti-shortcut implementation details", "private harness
     paths under tests/", "discoverable details that belong in local docs"]
  },
  "environment_changes": ["none, or small local evidence/dependency changes needed"],
```

```
  "rationale": "why the selected operator fits and why the rewrite is harder but
     controllable"
}

Rewrite target context:
<json.dumps(target_context, ensure_ascii=False, indent=2)>

Current seed task:
<json_task_context(task_dir)>
```

STEP 1: Oracle Solution Prompt

```
STEP 1: oracle solution protocol. Rewrite solution/solve.sh only.

Role:
Write the oracle shell implementation for the contract. The solution must complete the
     full workflow from the original initial state and behave like a strong terminal
     agent's successful trajectory.

Implementation requirements:
- Preserve the seed's main workflow and add the selected rewrite operator's requirement.
- Implement every stage in contract.task_chain in order: inspect, derive, execute,
     validate, finalize.
- Inspect inputs before transforming them; do not blindly overwrite final artifacts.
- Create or update every expected artifact with semantically meaningful content.
- Create observable intermediate/final artifacts needed by contract.reward_checks.
- Include bounded validation of intermediate artifacts before writing final outputs.
- Keep the script deterministic, idempotent where reasonable, and safe under repeated
     execution.
- Use local files, CLI tools, configs, fixtures, logs, or build/test commands that a
     terminal agent could discover.

Forbidden shortcuts:
- Hardcoding verifier-only constants without deriving or justifying them from
     discoverable evidence.
- Replacing the task with a no-op or a single trivial echo/write unless the seed itself
     demands that.
- Removing seed behavior to make the new verifier easier.
- Depending on internet access, external services, or unbounded background work.

Return schema:
{
  "status": "ok|blocked",
  "rationale": "short reason",
  "files": {"solution/solve.sh": "FULL FILE CONTENT"}
}

Task contract:
<json_contract_context(contract)>

Current task:
<json_task_context(task_dir)>
```

STEP 2: Verifier Prompt

```
STEP 2: verifier protocol. Allowed outputs are tests/test_state.py and tests/test.sh.

Role:
Write the oracle verifier for the contract, not for incidental implementation details in
      solution/solve.sh.

Verification requirements:
- Check the user-visible goal, preserved seed behavior, and all expected artifacts.
- Turn contract.reward_checks into distinct verifier subchecks where possible, using
     clear names that match the reward check names.
- Include at least four dense checks: evidence/discovery, intermediate artifact validity
     , final semantic correctness, and no-shortcut rejection.
- Verify semantic content/state, not only file existence.
- Enforce no-shortcut checks from contract.reward_checks.
- Preserve important seed-task checks unless the contract explicitly supersedes them.
- Prefer robust predicates over brittle command-order or exact implementation checks.
- Keep tests deterministic, fast, and compatible with Daytona sandbox execution.
```

```
Failure quality:
- Error messages should help diagnose missing artifacts, wrong formats, or inconsistent
    state.
- Do not leak a full solution recipe through assertion messages.

Return schema:
{
  "status": "ok|blocked",
  "rationale": "short reason",
  "files": {
    "tests/test_state.py": "FULL FILE CONTENT",
    "tests/test.sh": "FULL FILE CONTENT"
  }
}

Task contract:
<json_contract_context(contract)>

Current task:
<json_task_context(task_dir)>
```

STEP 3: Public Instruction Prompt (Part 1/2)

```
STEP 3: public instruction rewrite (discoverability-filtered). Rewrite instruction.md
    only.

Role:
Write instruction.md as a fair public user request for a terminal agent in Terminal-
    Bench / TerminalWorld style.
Solution already exists. Do NOT treat the verifier as a rubric to copy.

Example of desired style (do NOT copy content; match brevity):
Set up a Python project using Poetry in /app. Create a `.gitignore` containing
`test/inner_project/inner_project`, run `poetry install`, verify `inner_project`
imports in that environment, then use the project's console script to write a
valid JSON report at `/app/report.json` (see `--help` for arguments).

Critical anti-hacking rule:
- Dumping many absolute paths, schema fields, exact formats, or numbered operational
    steps into instruction.md makes agents shortcut / reward-hack.
- Prefer a compact goal + pointers to local docs. Put discoverable detail in workspace
    files, not in the instruction.

Path budget (match Terminal-Bench / TerminalWorld norms):
- Absolute paths are allowed, but usually only 1-2 (workspace entry like /app plus one
    main deliverable).
- At most ~3 absolute paths total. Never list path inventories of intermediate/meta
    artifacts.
- If more locations matter, point to a local README/spec/config instead of listing them.

Inputs you may use:
- current instruction
- oracle solution (workflow context only)
- a discoverability-filtered acceptance checklist
- the task contract boundary
Do NOT use or restate the full tests/ contents. The checklist already filtered them.

Fairness / discoverability (same policy as instruction repair):
- MUST include items under must_publish_in_instruction (missing from instruction AND not
     discoverable in workspace docs). Prefer the primary deliverable path over
    secondary ones if the list is long.
- For point_to_workspace_docs_instead: tell the agent to follow those files; do NOT
    restate their full contents.
- meta_artifacts_name_only: mention deliverable names if needed; no internal schemas.
- dropped_as_discoverable: do NOT dump these into the instruction.
```

STEP 3: Public Instruction Prompt (Part 2/2)

```
Style / restraint:
- Direct imperative, 1-3 short paragraphs. Prefer prose over bullets.
- No first-person chat ("I need you", "We need", ...).
```

```
- Keep close to current instruction length; at most ~1.4x current word count.
- Target about 80-130 words unless must-publish items require a little more.
- Do NOT expand field-level schemas for manifests/checksums/evidence.
- Do NOT enumerate every meta file's internal format.
- Do NOT paste solve.sh commands, anti-shortcut lines, or reward-check names.
- Do NOT write numbered walkthroughs / step checklists.

Hard bans:
- any path under /tests or reference to tests/test.sh, tests/test_state.py, local_check.
    sh under /tests;
- pytest / "run the tests" / "until tests pass" / "rerun until" / "the test suite fully
    specifies" as the acceptance loop;
- never tell the agent to open private harness files for acceptance criteria.

Use of the contract:
- Follow instruction_boundary.mention for public essentials.
- Keep instruction_boundary.hide out of the instruction.

Return schema:
{
  "status": "ok|blocked",
  "rationale": "short reason (fairness without rubric dump)",
  "files": {"instruction.md": "FULL FILE CONTENT"}
}

Task contract:
<json_contract_context(contract)>

Current instruction.md:
```
<instr>
```

solution/solve.sh (context only):
```
<solve>
```

Acceptance checklist (discoverability-filtered; NOT the full verifier):
<json.dumps(checklist, ensure_ascii=False, indent=2)>
```

STEP 4: Environment Alignment Prompt

```
STEP 4: environment alignment protocol.

Use the task contract as the source of truth.
Rewrite environment/Dockerfile and/or task.toml only if the contract requires a small
    environment metadata or dependency alignment.

Default preference: return an empty files object.

If you change environment files:
- preserve the seed's base image and installation style;
- keep the build context self-contained and ensure every COPY source exists;
- do not add internet-only runtime behavior;
- do not add proxies, secrets, credentials, or external service assumptions;
- keep storage, memory, and time requirements within normal Daytona limits;
- make only the minimum dependency or metadata change needed by the rewritten task.
- if the instruction intentionally hides exact constants, formats, paths, or checks,
    ensure those details are discoverable through existing local files or Dockerfile-
    created local files/scripts/fixtures.
- do not rely on the verifier as the only place where hidden details exist.
- do not introduce unguarded sed/perl edits against externally cloned dependency paths;
    prefer local fixtures or guard target files with test -f before editing.
- do not inflate resource requests or task timeout unless the existing seed already
    requires it.

Return schema:
{
  "status": "ok|blocked",
  "rationale": "short reason",
  "files": {
    "environment/Dockerfile": "FULL FILE CONTENT IF CHANGED",
    "task.toml": "FULL FILE CONTENT IF CHANGED"
  }
```

```
}

Task contract:
<json_contract_context(contract)>

Current task:
<json_task_context(task_dir)>
```

STEP 5: Consistency Repair Prompt

```
STEP 5: consistency repair protocol.

Fix cross-file inconsistencies so the oracle solution should pass the verifier in a
     fresh sandbox build.

Mandatory audit procedure:
1. List every file/path the verifier tests require to exist or contain specific content.
2. List every file/path solution/solve.sh creates or updates.
3. List every file/path promised in contract.expected_artifacts and every artifact/state
      implied by contract.task_chain.
4. List every dense reward check in contract.reward_checks and identify which verifier
     subcheck enforces it.
5. Compare these lists; any verifier-required path or reward-check artifact missing from
      the solution plan is a bug.
6. Mentally execute solution/solve.sh in order; if a step would fail or skip an artifact
     , fix the solution.
7. If environment/Dockerfile is supposed to provide config/scripts/fixtures, confirm
     they are actually created in the Dockerfile.
8. If instruction points to a path, config, validation script, README, fixture, or log,
     confirm that evidence exists or is created by the environment.
9. Instruction fairness audit:
   - If instruction mentions /tests, tests/test.sh, pytest acceptance loops, or "until
     tests pass", remove those leaks.
   - Prefer fixing unfair gaps by adding/repairing workspace evidence (README/spec/
     config) that instruction already points to - not by dumping verifier paths/fields/
     steps into instruction.md.
   - Only publish an exact string/path/completion constant in instruction.md when it is
     undiscoverable from workspace evidence.
   - Do not weaken verifier checks to hide unfair gaps.
   - Do not turn instruction into a verifier dump, schema inventory, or numbered
     walkthrough (reward hacking).

Repair only genuine consistency problems. Do not change the selected rewrite_family/
     rewrite_operator or invent a new task.
Prefer fixing solution/environment/instruction mismatches over weakening verifier checks
     .
Keep instruction.md compact; expand only for a few undiscoverable hard constants.
Return full file contents only for changed files. Do not create new files outside this
     whitelist:
solution/solve.sh, tests/test_state.py, tests/test.sh, instruction.md, environment/
     Dockerfile, task.toml.

Return schema:
{
  "status": "ok|blocked",
  "rationale": "short reason",
  "verifier_required_paths": ["paths or artifacts checked by verifier"],
  "solution_created_paths": ["paths or artifacts created by solution"],
  "contract_promised_paths": ["paths or artifacts promised by contract.
     expected_artifacts/task_chain/reward_checks"],
  "issues_found": ["specific cross-file inconsistency, or empty list"],
  "repairs_made": ["specific repair made, or empty list"],
  "files": {
    "solution/solve.sh": "FULL FILE CONTENT IF CHANGED",
    "tests/test_state.py": "FULL FILE CONTENT IF CHANGED",
    "tests/test.sh": "FULL FILE CONTENT IF CHANGED",
    "instruction.md": "FULL FILE CONTENT IF CHANGED",
    "environment/Dockerfile": "FULL FILE CONTENT IF CHANGED",
    "task.toml": "FULL FILE CONTENT IF CHANGED"
  }
}

Task contract:
<json_contract_context(contract)>
```

```
Current task:
<json_task_context(task_dir)>
```

DAYTONA FEEDBACK REPAIR Prompt

```
DAYTONA FEEDBACK REPAIR.

The task has already been generated and then run in Daytona oracle validation.
Use the dynamic failure logs below as the source of truth. Repair only the files needed
     so solution/solve.sh passes the verifier in Daytona.

Rules:
- If result.json shows build failure, repair environment/Dockerfile or task.toml.
- If verifier says an artifact is missing, make solution/solve.sh create it, unless the
     artifact should be provided by environment.
- If verifier says content is malformed/empty, fix solution generation logic.
- If verifier and contract disagree, minimally repair the verifier to match the contract
     .
- Do not weaken verifier just to pass; preserve anti-shortcut checks.
- Keep instruction compact. Do not copy Daytona verifier failures, hidden expected
     values, complete artifact lists, schema fields, reward checks, or exact assertions
     into instruction.md.
- Do not expand instruction.md during repair unless it is inconsistent or impossible to
     start from; if editing it is necessary, keep it as a public request with at most
     two concrete entry points.
- Return full file contents only for changed files.
- Do not create new files. Do not return any path outside this whitelist:
  solution/solve.sh, tests/test_state.py, tests/test.sh, instruction.md, environment/
     Dockerfile, task.toml.

Return schema:
{
  "status": "ok|blocked",
  "failure_type": "build_failed|solution_artifact_missing|solution_content_wrong|
     verifier_mismatch|environment_missing_evidence|other",
  "root_cause": "specific reason based on Daytona logs",
  "repairs_made": ["specific file-level repairs"],
  "files": {
    "solution/solve.sh": "FULL FILE CONTENT IF CHANGED",
    "tests/test_state.py": "FULL FILE CONTENT IF CHANGED",
    "tests/test.sh": "FULL FILE CONTENT IF CHANGED",
    "instruction.md": "FULL FILE CONTENT IF CHANGED",
    "environment/Dockerfile": "FULL FILE CONTENT IF CHANGED",
    "task.toml": "FULL FILE CONTENT IF CHANGED"
  }
}

Task contract:
<json_contract_context(contract)>

Current task files:
<json_task_context(task_dir)>

Daytona failure evidence:
<json.dumps(evidence, ensure_ascii=False, indent=2)>
```

G Task-Quality Comparison: Recursive versus Single-Pass Rewriting

To compare recursive rewriting with a single rewrite, we evaluate RST rounds against two single-pass baselines that share the same rewrite-operator taxonomy, environment scaffolding, and validation pipeline but stop after one rewrite step. The instruction-first single-pass baseline generates a new public instruction first and then derives a matching solution and verifier, while the solution-first single-pass baseline mirrors the RST order by generating a new reference solution first and then aligning the verifier and instruction to it. Both baselines start from the same 639 bootstrap seeds used by R_1. From each of the four RST rounds (R_1, R_5, R_{10}, R_{15}) and the two single-pass baselines, we randomly sample 100 tasks that pass the standard synthesis pipeline and audit them with two separate DeepSeek-V4-Pro reviewer calls, using a third DeepSeek-V4-Pro call to adjudicate disagreements. Each accepted task receives a 1–5 score along seven release-quality dimensions —

instruction clarity, instruction self-containedness, instruction–verifier alignment, verifier fairness, verifier coverage, solution correctness, and solution reasonableness — and a separate 1–5 task-value score for whether the task represents useful, non-degenerate terminal-agent work. Release readiness is the mean of the seven release-quality dimensions. We define an LLM-rated utility proxy as $0.4 \cdot \text{release readiness} + 0.6 \cdot \text{task value}$; this weighted reviewer score is not a direct measurement of downstream training performance.

Table 12 reports the resulting 1–5 means over the 100 randomly sampled tasks from each source. The LLM-rated utility proxy increases monotonically with RST round index, from 4.514 at R_1 to 4.761 at R_{15}, and every RST round strictly exceeds both single-pass baselines (4.377 and 4.430). Task value likewise increases from 4.325 at R_1 to 4.850 at R_{15}, compared with 4.237–4.300 for the two single-pass sources. Release readiness follows a different pattern: it peaks at R_1 (4.796) and declines gradually to 4.629 at R_{15}, which remains comparable to the two single-pass baselines (4.586 and 4.625).

The per-dimension results are correspondingly mixed. RST R_1 exceeds both single-pass baselines on all seven release-quality dimensions. At R_5, instruction self-containedness and solution correctness are below the stronger single-pass value; at R_{10}, instruction clarity and verifier fairness are lower; and at R_{15}, instruction–verifier alignment, solution correctness, and solution reasonableness trail the strongest single-pass result. Nevertheless, RST R_{15} remains above both single-pass baselines on verifier coverage and attains the highest task-value and LLM-rated utility-proxy scores among all sources. Overall, the proxy increases monotonically while release readiness remains competitive, suggesting that later rounds increase judged task value without uniformly improving every release-quality dimension.

Table 12: Task-quality audit comparing two single-pass baselines against four RST rounds. Each column summarizes 100 randomly sampled tasks from the corresponding source. Rows are audit dimensions; columns are task sources. All numbers are 1–5 means from separate DeepSeek-V4-Pro reviewer calls, with a third call adjudicating disagreements. Release readiness is the mean of the seven upper dimensions; the LLM-rated utility proxy is $0.4 \cdot \text{release readiness} + 0.6 \cdot \text{task value}$ and is not a direct downstream-training measurement.

Dimension	Instruction-first single-pass	Solution-first single-pass	RST R_1	RST R_5	RST R_{10}	RST R_{15}
Instruction clarity	4.400	4.525	4.700	4.725	4.471	4.550
Instruction self-containedness	4.562	4.500	4.737	4.500	4.676	4.750
Instruction–verifier alignment	4.612	4.713	4.938	4.775	4.735	4.700
Verifier fairness	4.388	4.425	4.650	4.525	4.176	4.450
Verifier coverage	4.662	4.725	4.925	4.875	4.853	4.800
Solution correctness	4.800	4.763	4.812	4.775	4.824	4.650
Solution reasonableness	4.675	4.725	4.812	4.800	4.824	4.500
Task value	4.237	4.300	4.325	4.700	4.794	4.850
Release readiness	4.586	4.625	4.796	4.711	4.651	4.629
LLM-rated utility proxy	**4.377**	**4.430**	**4.514**	**4.704**	**4.737**	**4.761**

Blinded human quality audit. From each single-pass baseline and audited RST round, we randomly sample 50 accepted tasks, hide their source and round, and have two annotators independently inspect the complete bundles; a third adjudicates disagreements. Three binary criteria capture whether every reward-relevant condition is stated or discoverable (*publicly solvable*), the verifier tests the stated task without hidden or contradictory requirements (*instruction–verifier aligned*), and both conditions hold without a critical correctness or degeneracy defect (*overall release-ready*).

Table 13 shows that R_1 has the highest pass rate on all three criteria. Later RST rounds retain 90–92% public solvability and 86–88% release readiness, above the single-pass ranges of 86–88% and 82–84%. Alignment declines from 96% in R_1 to 88% in R_{15}, placing R_{15} between the two single-pass baselines.

Table 13: Blinded human audit of 50 accepted tasks per source. Two annotators independently inspect each complete task bundle with its source and round hidden; a third adjudicates disagreements. The three quality columns report adjudicated binary pass rates. Raw agreement is task-level exact agreement between the two initial annotators across all three criteria; the remaining tasks enter adjudication.

Source	N	Publicly solvable (%)	Instruction–verifier aligned (%)	Overall release-ready (%)	Raw exact agreement	Adjudication rate (%)
Instruction-first single-pass	50	86	84	82	38/50 (76%)	24
Solution-first single-pass	50	88	90	84	39/50 (78%)	22
RST R_1	50	94	96	92	42/50 (84%)	16
RST R_5	50	92	90	88	40/50 (80%)	20
RST R_{10}	50	90	90	88	41/50 (82%)	18
RST R_{15}	50	90	88	86	42/50 (84%)	16

Table 14: Downstream benchmark evaluation configurations. All three benchmarks use temperature 0.6, top-p 0.95, a 131,072-token context window, an 8,192-token single-generation limit, non-thinking mode, and SGLang serving in bfloat16 (bf16) with tensor parallelism 4 and data parallelism 2.

Benchmark	Additional decoding	Horizon and token limits	Timeout and concurrency
Terminal-Bench 2	Top-k 20	100 turns; 120,000 maximum input tokens	Original task timeout $\times 6$; 89 concurrent tasks per node
Terminal-Bench Hard	Top-k 20	100 turns; 120,000 maximum input tokens	Original task timeout $\times 6$; evaluation concurrency 100
Long-Horizon Terminal Bench	No explicit top-k or stop sequences	1,000 turns; 120,000/8,192 maximum input/output tokens	5,400 s per task; task and setup timeout multipliers 1; 46 concurrent tasks per node; `continue_until_timeout=true`

In particular, the 86% release-readiness rate at R_{15} means that 14% of automatically accepted tasks still contain at least one critical correctness, alignment, or degeneracy defect under human review. Automated acceptance checks execution but does not guarantee semantic release readiness.

Task-level exact agreement between the two initial annotators ranges from 76% to 84%; pooled across all 300 audited tasks, 242 receive identical three-criterion labels (80.7%) and 58 require adjudication (19.3%).

H Training and Evaluation Implementation

The trajectory counts correspond to distinct stages of the data pipeline. The 327,189 trajectories in Table 8 count all recorded agent-interaction traces and are not the size of one SFT training set.

The controlled ablation in Table 1 trains on 1,000 examples from each of the corresponding R_1–R_3 conditions (3,000 total) and runs the complete training and evaluation independently with seeds 42, 43, and 44. In table-row order, the underlying Terminal-Bench 2 solved-task counts for these seeds are 38/36/37, 38/36/39, 38/39/38, and 40/40/41 out of 89.

The stage-wise experiment in Section 4.5 is sequential and model-specific: after removing trajectories associated with duplicate task records, each stage retains 10,000 successful trajectories generated by the corresponding model. For each of Qwen3.5-27B and Qwen3.5-122B-A10B and each training-and-evaluation seed (42, 43, or 44), the base checkpoint generates the Round 1 rollouts and is fine-tuned on the retained R_1 set. For Rounds 2 and 3, the checkpoint updated in the preceding stage generates rollouts on the corresponding R_2 or R_3 tasks. That checkpoint is then fine-tuned on the newly retained trajectories. Sequential SFT uses only the first three rounds because rollout success rates decline in later rounds, increasing the cost of collecting successful trajectories.

The fixed-token-budget experiment in Section 4.6 is separate and directly samples 11,531 unique Qwen3.5-27B trajectories at random from the pooled R_1–R_3 data before training and evaluation with the same three run seeds.

Table 15: Training and evaluation hardware. All GPU stages use NVIDIA H20.

Stage	Hardware
SFT trajectory rollouts (Qwen3.5-27B)	3 nodes × 8 H20; SGLang TP4/DP2; bf16; 131,072-token context
SFT training (27B and 122B)	8 nodes × 8 H20 (64 GPUs); SLIME / Megatron
PPO training (27B)	Same 64-GPU H20 stack as 27B SFT
Downstream SGLang evaluation	1 node × 8 H20 per concurrent eval job; TP4/DP2; bf16
Oracle validation and Harbor trials	Daytona sandboxes; no local GPU

For Qwen3.5-27B, SFT trajectories are generated by running agent rollouts on TerminalWorld-style tasks with SGLang. The base checkpoint is used for the first round, and the checkpoint updated in the preceding SFT stage is used for each subsequent round. Rollout decoding uses temperature 1.0 and top-p 0.95, with at most 100 interaction turns and 8,192 generation tokens per model call. Rollouts execute on three H20 nodes with eight GPUs each, using tensor parallelism 4, data parallelism 2, bfloat16 (bf16) precision, and a 131,072-token context window. Each node executes 10 rollout loops with up to 200 concurrent Daytona trials, and complete trajectory records are retained for SFT conversion. Qwen3.5-122B-A10B follows the same schedule with its own base and updated checkpoints.

The base checkpoints are the official Hugging Face releases `Qwen/Qwen3.5-27B` and `Qwen/Qwen3.5-122B-A10B`. SFT uses the SLIME training framework with a Megatron backend on 64 GPUs. TP, DP, PP, CP, and EP below denote tensor, data, pipeline, context, and expert parallel sizes, respectively. For Qwen3.5-27B, each 10,000-trajectory stage trains for one epoch with global batch size 128, context length 262,145, Qwen-specific loss masking, TP4/PP2/CP2 parallelism, and cosine learning-rate decay from 3×10^{-6} to 3×10^{-7}.

Qwen3.5-122B-A10B uses the same 64-GPU SLIME stack with mixture-of-experts-specific TP2/PP8/CP2/EP4 parallelism, data-parallel size 2, sequence parallelism, and six layers on the last pipeline stage. Each 10,000-trajectory stage uses global batch size 8, context length 262,145, a 49,152-token per-GPU packing limit, Qwen-specific loss masking, sample-level SFT loss, and cosine learning-rate decay from 1×10^{-6} to 1×10^{-7} with 5% warmup and no weight decay. Hopper Triton gated-attention backward kernels are disabled for this MoE recipe.

PPO configuration. The Qwen3.5-27B actor starts from the base checkpoint, while a separate full-size critic is warm-started from a prior terminal-agent critic trained on the remaining TMax data after excluding all 100 Terminal-Bench Hard evaluation tasks. The critic uses fresh optimizer state, and two critic-only steps precede actor updates. Tasks are sampled uniformly from the 37,484 accepted RST tasks, with one rollout per task. Each step launches 520 trials and retains the first 512 completed trajectories. Rollouts use temperature 1.0, top-p 1.0, no top-k truncation, at most 120 tool-call turns, a 4,096-token per-turn generation limit, an 86,016-token context, and non-thinking mode.

For P passing verifier assertions and binary completion b, the reward is $r = P/20$ when the per-assertion report is available and $r = b$ otherwise. There is no completion bonus, failure or length penalty, reward clipping, or reward normalization, so rewards can exceed one. PPO uses Adam with $\beta = (0.9, 0.98)$, weight decay 0.1, constant actor/critic learning rates of $2 \times 10^{-6}/1 \times 10^{-5}$, policy and value clipping of 0.2, zero Kullback–Leibler (KL) and entropy coefficients, generalized advantage estimation (GAE) with $\gamma = \lambda = 1$, and batch-level advantage whitening. One PPO epoch is applied to each 512-trajectory batch. Three independent runs use training-and-evaluation seeds 42, 43, and 44; each performs 70 steps (35,840 retained trajectories) and is evaluated at its final step-69 checkpoint. Their mean and standard deviation therefore include both training- and evaluation-run variation.

Downstream checkpoints are served with SGLang and evaluated with Harbor and Daytona under Table 14. SFT and PPO statistics summarize independent training-and-evaluation runs with seeds 42, 43, and 44 and include training-run variation; base rows use the same three evaluation seeds without post-training. Fixed solvers use the same task and sandbox protocol through an external-API proxy. The GPT endpoint is `gpt-5.6-sol-2026-07-09`; both solvers use provider defaults with thinking disabled.

Hardware, API, decoding, and snapshot pins. Table 15 records the compute used for synthesis rollouts, SFT, PPO, and downstream evaluation. Table 16 pins the generation and evaluation APIs together with the local Harbor and Daytona versions used by the evaluation entrypoints. Table 17 collects the decoding settings that are not already listed in Table 14.

Table 16: API identifiers and software versions. DeepSeek generation uses the DataEval proxy; GPT evaluation uses the dated solver endpoint.

Component	Identifier / version	Role
DeepSeek generation	`api_deepseek_deepseek-v4-pro` via DataEval	Recursive synthesis, repair, and LLM quality audit
GPT fixed solver	`gpt-5.6-sol-2026-07-09`	Fixed-solver difficulty evaluation
Qwen3.5-27B Base	`Qwen/Qwen3.5-27B` `fc05daec18b0a78c049392ed2e771dde82bdf654`	SFT / PPO initialization
Qwen3.5-122B-A10B Base	`Qwen/Qwen3.5-122B-A10B` `dc4d348443bc740c68e2d77492492c11606384d5`	MoE SFT initialization
Harbor (eval entrypoint)	0.7.0, agent `terminus-2`	Downstream and oracle Harbor runs
Daytona SDK	0.211.2	Sandbox validation and evaluation
Training seeds	42, 43, 44	Independent SFT / PPO / evaluation repeats

Table 17: Decoding parameters by stage. Downstream benchmark settings in Table 14 are repeated here only where they differ from rollout or API defaults.

Stage	Temperature	Nucleus / top-k	Horizon and token limits
Task synthesis / repair	0.2	top-p 0.95	900 s request timeout; $\leq$ 2 retries
SFT rollouts (27B)	1.0	top-p 0.95	100 turns; 8,192 gen. tokens; 131,072 context
PPO rollouts (27B)	1.0	top-p 1.0; no top-k	120 tool-call turns; 4,096 gen. tokens; 86,016 context
TB2 / TB-Hard eval	0.6	top-p 0.95; top-k 20	100 turns; 8,192 gen.; 120,000 input; non-thinking
LHTB eval	0.6	top-p 0.95; no top-k	1,000 turns; 5,400 s; `continue_until_timeout`
GPT / DeepSeek eval proxy	provider default	provider default	thinking disabled