LEARNING GENERALIZABLE BEHAVIORS FOR TERMINAL AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Terminal agents have emerged as a compelling application of large language models (LLMs), with the potential to integrate deeply into users' daily workflows. Reinforcement learning (RL) has become a key technique for improving their capabilities, making scalable training environments a central challenge. Since public real-user interaction data are scarce, synthetic environments provide a practical alternative, but often exhibit substantial domain gaps and limited fidelity, resulting in poor generalization. Existing efforts primarily focus on scaling the quantity and diversity of synthetic environments, while reward signal quality and the mechanisms governing generalization remain underexplored. In this work, we study how RL improves terminal agents. We propose and validate the **agentic compositional generalization** hypothesis: rather than teaching the agent new domain-specific skills from scratch, RL primarily shapes high-level decision-making behaviors that compose and route low-level skills acquired during pre-training and supervised fine-tuning (SFT). This perspective suggests that verifier quality, which determines the behaviors reinforced during RL, is more important than simply increasing the number or diversity of training environments. Motivated by this insight, we propose RIVER, a simple training recipe that improves the reward signal by filtering low-quality environments and augmenting the outcome reward with process-level behavior regularization. Using this recipe, our RL-trained agent achieves the best performance among evaluated open-source RL-trained 8B models across four terminal-agent benchmarks and generalizes across model families, scales, agent harnesses, and RL objectives. Using **fewer than 30%** of the TMax training environments, RIVER improves RL gains by **106% and 30%** on average for models ranging from 2B to 27B on Terminal-Bench-Lite and Terminal-Bench-v2.1, respectively.

1 INTRODUCTION

As large language models (LLMs) continue to advance toward real-world deployment, *terminal agents* have emerged as one of their most important applications, enabling software engineering, system administration, and other interactive coding tasks (Anthropic, 2025; OpenAI, 2025; xAI, 2025; Pi et al., 2026; Ivison et al., 2026; Wu et al., 2026b). While the capabilities of frontier terminal agents have improved rapidly, the principles underlying the design of their training environments remain largely undisclosed (Pi et al., 2026; Ivison et al., 2026; Zhu et al., 2026). In particular, it remains unclear how post-training for terminal agents generalizes across domains, and what factors matter beyond simply increasing coverage over task categories and targeted skills.

Reinforcement learning (RL) has become the dominant paradigm for post-training reasoning models and agents (OpenAI, 2024; Guo et al., 2025). Unlike single-turn reasoning tasks such as mathematical reasoning (Shao et al., 2024), terminal agents require interacting with executable environments through long-horizon, multi-turn decision-making (Gandhi et al., 2026). Consequently, the training environment plays a central role in agentic RL. Since large-scale real-user interaction data are generally unavailable, recent work has increasingly relied on *synthetic environments* as a scalable alternative (Pi et al., 2026; Li et al., 2026).

Existing work on synthetic terminal environments (Pi et al., 2026; Fan et al., 2026; Zhao et al., 2026) has primarily focused on *scaling coverage*: expanding the diversity of tasks, domains, and re-

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.

quired skills. Early efforts generated demonstration trajectories for supervised fine-tuning (SFT), whereas more recent work constructs executable environments for RL. For example, Endless-Terminals (Gandhi et al., 2026) demonstrated that RL over synthetic environments substantially improves terminal agents, and TMax (Ivison et al., 2026) further increased environment diversity and domain coverage [1]. However, *coverage alone does not explain generalization* (see Section 4.4). This raises two fundamental questions: *(i)* What does RL actually learn during terminal-agent training? *(ii)* Beyond diversity, what makes one set of training environments more valuable than another for generalizing beyond the synthetic training distribution?

In this work, we study these questions through the lens of the *compositional generalization* hypothesis (Yuan et al., 2026; Kong et al., 2026), which holds that RL for single-turn reasoning mainly learns to compose previously acquired skills. We extend this view to multi-turn, long-horizon agentic agents. We term the resulting hypothesis, *agentic compositional generalization* (see Fig. 1 for an overview), and test it in realistic terminal environments rather than synthetic tasks. Our findings indicate complementary roles of SFT and RL in generalization: SFT (and general model prior imposed by pre-training and mid-training) primarily equips the agent with low-level atomic skills, while RL primarily shapes *high-level decision-making behaviors* that compose, route, and coordinate these skills across domains. Under this hypothesis, generalization requires both coverage on atomic skills and reusable high-level behaviors.

This perspective suggests a different axis for the design of the training recipe. For skills, since RL is not efficient in learning atomic skills, SFT should equip the models with diverse atomic skills that are necessary for target domains. For RL-learned behaviors, *verifiers* in training environments, which provide the signal that shapes those behaviors, become critical. Guided by this insight, we propose RIVER, a simple yet effective training recipe that filters out low-quality environments and improves verifier quality through behavior-level regularization. Despite using fewer than 30% of the TMax training environments, RIVER consistently improves RL performance across model families, model scales, agent harnesses, and RL algorithms. Our contributions are summarized as follows:

(1) **Mechanistic interpretation.** We propose *agentic compositional generalization*, which posits that RL improves generalization by shaping reusable high-level behaviors that compose atomic skills acquired before RL, and we present empirical evidence consistent with its predictions.

(2) **A simple and effective training recipe.** We introduce RIVER, which improves the quality of the reward signal and the training efficiency. Using fewer than 30% of the TMax environments (Ivison et al., 2026), RIVER achieves on average 106% and 30% larger RL gains on Terminal-Bench-Lite and Terminal-Bench-v2.1, respectively, across models ranging from 2B to 27B.

(3) **Comprehensive empirical analysis.** We provide extensive experiments that test the predictions of our hypothesis and offer new insights into the learning dynamics of RL for terminal agents.

2 MECHANISM INTERPRETATION AND METHOD

To explain how terminal agents generalize beyond synthetic training tasks, we propose the **Agentic Compositional Generalization** (**ACG**) hypothesis. It motivates **R**eward-**I**ntegrity **V**erified **E**nvironments for **RL** (**RIVER**), a training recipe combining broad skill coverage with verifier-aware environment filtering and behavior-level regularization.

2.1 AGENTIC COMPOSITIONAL GENERALIZATION

We formalize terminal agent operations as a Markov Decision Process (MDP) (Puterman, 2014). At step t, state s_t contains the complete interaction history, including the user instruction, executed commands, and terminal feedback. Action a_t is the agent's single-turn output, including reasoning tokens and any tool invocation or terminal command; policy $\pi(a_t \mid s_t)$ defines its action distribution. Within this formulation, we distinguish two complementary levels of competence (Fig. 1):

Low-level Skills. A skill is the ability to execute action a_t correctly in state s_t, using domain knowledge, reasoning, and appropriate tools (e.g., composing a valid grep pipeline; see App. C.1 for more examples). Skills concern local correctness irrespective of the surrounding strategy. SFT primarily improves skills by concentrating $\pi(\cdot \mid s_t)$ on valid, domain-appropriate actions.

[1] We provide more discussion on related works in App. A.

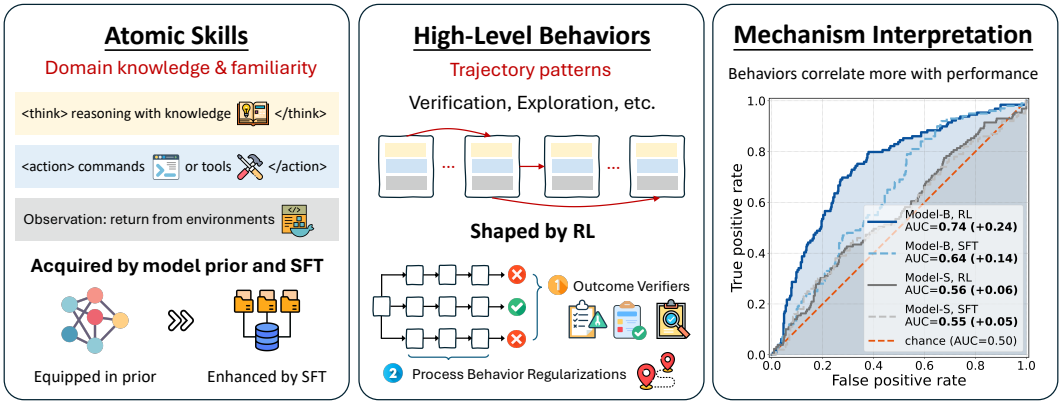


Figure 1: The **Agentic Compositional Generalization** hypothesis. (**Left**) Pre-training and SFT provide low-level skills for executing individual actions. (**Middle**) Environment feedback during RL shapes behaviors that compose and route these skills across turns. (**Right**) Behavior features predict task success more strongly than skill presence (AUC-ROC 0.74 vs. 0.55), with the association strengthening from SFT to RL (0.64 to 0.74). See Sec. 4.2 for analysis details.

High-level Behaviors. A behavior is a pattern across the state-action pairs (s_t, a_t) induced by π, typically spanning multiple turns, such as verifying results before declaring completion (details in App. E.2). Behaviors determine how actions are selected and coordinated across turns.

We propose the **Agentic Compositional Generalization** hypothesis, extending recent theories of compositional generalization for single-turn reasoning models (Kong et al., 2026; Yuan et al., 2026) to multi-turn terminal agents. The main idea is that the agent generalizes by combining and re-routing behaviors learned and shaped during RL with the domain skills required by the target task. In this section, we present the intuition and implications of this hypothesis, while deferring the empirical validation to Sec. 4, where we provide comprehensive experimental evidence supporting these claims.

Roles of SFT. Our findings suggest that SFT plays a critical role by familiarizing the model with the target domains and interaction harness, enabling it to produce sufficient correct trajectories to yield meaningful learning signals during RL if these capabilities were not well-initialized in pretraining. More importantly, SFT equips the model with the low-level skills required for downstream tasks, providing the foundation upon which RL learns high-level behaviors that, under our hypothesis, enable generalizability.

Roles of RL. Based on our experiments and analysis, RL primarily reshapes *high-level behaviors* that "re-route" low-level skills. We hypothesize that this behavioral reorganization is an important driver of generalizability. As shown in Fig. 1, an AUC-ROC analysis (Fawcett, 2006) demonstrates that behavior features are critical and predictive of task success (see details in Sec. 4.2). Thus, providing appropriate signals in agentic RL to foster **generalizable behaviors** and reduce reward hacking is important to terminal agent training.

2.2 RIVER: REWARD-INTEGRITY VERIFIED ENVIRONMENTS FOR RL

Our hypothesis suggests two complementary training requirements: broad atomic-skill coverage and reliable feedback for shaping behaviors. For SFT, we use a filtered OpenThoughts-SFT corpus (Raoof et al., 2026) to provide broad skill coverage (see Sec. 4.3 and App. B.1 for details). For RL, as shown in Fig. 2, each synthetic terminal environment provides a sandbox, task-specific files, and outcome verifiers (e.g., test cases). Inaccurate verifiers can produce inaccurate rewards, leading to reward hacking (App. D.4). RIVER improves the quality of reward signals in the environment set through *environment filtering* and *verifier enhancement*, as highlighted in Fig. 2.

Environment filtering. Although synthetic environments have become a dominant paradigm for terminal-agent RL, their quality remains highly variable. Existing open-source collections often contain weak or mismatched verifiers, resulting in the potential for noisy supervision that encourages shortcut solutions, reward hacking, or overfitting. Since verifier quality directly shapes the high-level behaviors learned during RL, ensuring environment quality is a prerequisite for effective training.

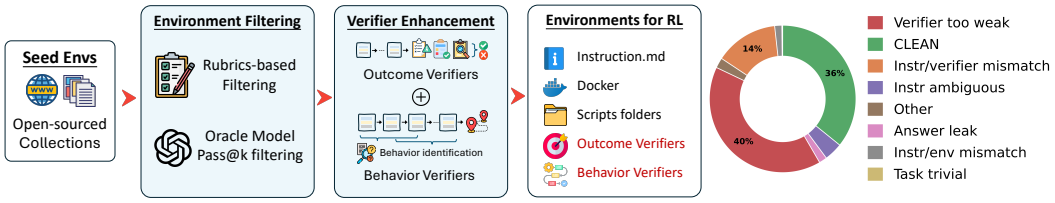


Figure 2: (**Left:**) Pipeline for environment filtering and verifier enhancement. (**Right:**) Environment quality evaluation results of TMax-15K (Ivison et al., 2026).

We develop a lightweight environment filtering system that evaluates synthetic environments using a set of rubrics without executing the underlying sandbox. Our framework identifies seven categories of quality issues, including weak verifiers, mismatched task specifications, and other inconsistencies (see App. D for details and examples). Applying this evaluation to the TMax (Ivison et al., 2026) environment collection reveals that more than 60% of the environments exhibit at least one quality issue[2] (see the right figure of Fig. 2).

We further filter the remaining environments using GPT-5.4 (OpenAI, 2026) with the harness of Terminus-2 (Harbor Framework Team, 2026) as an oracle verifier. Specifically, we perform a pass@$k = 2$ evaluation and remove environments with zero pass rate, which are either excessively difficult or incompatible with our training infrastructure. The resulting curated collection, **RIVER-TMax-3.5K**, serves as the training environment set throughout this work.

Verifier enhancement. Environment filtering improves task quality but leaves sparse outcome supervision. Our *verifier enhancement* complements outcome rewards with lightweight behavior verifiers, providing turn-level feedback to encourage generalizable interaction patterns while remaining compatible with standard RL objectives.

Algorithm 1: Full Training recipe.

(1) SFT dataset construction with broad skill coverage and SFT training # Details in App. B.1, Discussions in Sec. 4.3

RIVER core components:
(2) RL Environment Filtering
(2.1) Deploy Rubrics-based Filtering;
(2.2) Oracle Agent Pass@k Filtering;
(3) Verifier Enhancement (1)

(4) RL on final envs

We instantiate this in GRPO-style objective (Shao et al., 2024). For each prompt, we sample G rollouts and, unless otherwise stated, assign binary rewards $r_i \in \{0, 1\}$, with $r_i = 1$ only if rollout i passes all verifiers and test cases. We normalize rewards within the group and broadcast the scalar advantage to every response token. To provide behavior-level feedback, we add a turn-wise shaping term $s_{i,t}$:

$$A_{i,t} = \frac{r_i - \bar{r}}{\sigma_r + \epsilon}, \qquad \bar{r} = \frac{1}{G}\sum_{j=1}^{G} r_j, \qquad \sigma_r = \mathrm{std}(\{r_j\}_{j=1}^{G}), \qquad \tilde{A}_{i,t} = A_{i,t} + s_{i,t} \tag{1}$$

Here, $\bar{r}$ and σ_r are the within-group reward mean and standard deviation, and ϵ is a small constant. Term $s_{i,t}$ can be positive to reinforce desirable behaviors or negative to discourage undesirable ones. We penalize repetitive loops with little information gain using $s_{i,t} = -\delta\,\mathbf{1}[t \in \bigcup_{k\in\mathcal{L}_i} \mathcal{T}(k)]$, where $\mathcal{T}(k)$ contains turn k's response-token indices and $\mathcal{L}_i$ contains turns whose command and observation each have Jaccard similarity greater than 0.8 with an earlier turn. We set $\delta = 0.05$ in the implementation. Details and additional behavior-verifier experiments are provided in Sec. 3.1.

Recipe summary and intuition. Our overall training recipe is summarized in Alg. 1. Additional implementation details are provided in Sec. 3 and App. B. The central intuition is simple: make the model RL-ready by equipping it with sufficient low-level skills, and make the high-level behaviors learned through RL generalizable by providing high-quality verifier signals from environments.

3 EXPERIMENT RESULTS

In this section, we evaluate the RIVER pipeline using Qwen3, Qwen3.5, and Qwen3.6 models (Yang et al., 2025; Team, 2026a;b). Evaluation is conducted on four terminal benchmarks:

[2]Our filtering process may produce both false positives and false negatives. Also, there might be multiple issues in one environment. Details of environment filtering can be found in App. D. Examples of reward hacking with unfiltered environments can be found in App. D.4.

Table 1: Evaluation results of 8B agents on four terminal benchmarks. Numbers are benchmark scores ($\times 10^2$). **Avg.** is the unweighted mean of the four benchmarks. Scores are mean $\pm$ std over 3 seeds. Baseline scores are reported by our evaluation pipeline under fair comparison.

Model	Harness	Training	Otbl	Tbv2.1	Tbpro	Twv	**Avg.**
River-8B (ours)	EndlessAgent	SFT+RL	**21.8** $\pm$2.9	**9.7** $\pm$0.6	**23.0** $\pm$3.1	**23.0** $\pm$2.5	**19.4** $\pm$1.2
OpenThinker-8B-RL	Terminus-2	SFT+RL	18.4 $\pm$1.5	8.6 $\pm$0.6	21.8 $\pm$1.6	22.3 $\pm$1.5	17.8 $\pm$0.7
OT-SFT-Endless-8B	EndlessAgent	SFT+RL	14.1 $\pm$3.6	6.4 $\pm$1.7	16.8 $\pm$0.8	21.7 $\pm$1.9	14.7 $\pm$1.1
TMax-RL-8B	Vanillux2	RL	17.0 $\pm$3.7	6.0 $\pm$0.6	16.3 $\pm$1.0	19.0 $\pm$2.6	14.6 $\pm$1.2
OpenThinker-Agent-v1	Terminus-2	SFT+RL	16.0 $\pm$0.6	5.2 $\pm$0.6	16.2 $\pm$1.0	21.7 $\pm$2.9	14.8 $\pm$0.8
TMax-SFT-8B	Vanillux2	SFT	12.1 $\pm$3.4	4.5 $\pm$1.9	13.7 $\pm$1.9	14.3 $\pm$4.1	11.2 $\pm$1.5
River-SFT-8B	EndlessAgent	SFT	11.3 $\pm$2.3	6.4 $\pm$1.3	13.7 $\pm$2.6	18.5 $\pm$2.3	12.5 $\pm$1.1
Qwen3-8B	EndlessAgent	None	3.8 $\pm$1.3	2.2 $\pm$1.2	6.8 $\pm$0.6	9.8 $\pm$1.6	5.7 $\pm$0.6

(OpenThoughts-)Terminal-Bench-Lite (Raoof et al., 2026), **Terminal-Bench-v2.1** (Merrill et al., 2026), **Terminal-World-Verified** (Chu et al., 2026), and **Terminal-Bench-Pro** (Wang et al., 2025b). For brevity, these benchmarks are referred to as `Otbl`, `Tbv2.1`, `Twv`, and `Tbpro`, respectively, when there is no ambiguity. In addition, a validation subset of approximately 300 tasks is held out from the TMax collection, excluded from training, and used for controlled experiments. Each benchmark is evaluated under its default criterion: `Otbl` uses a continuous score in $[0, 1]$, whereas the remaining benchmarks use binary scores, assigning 1 only when the solution passes all verifiers.

3.1 QWEN3-8B MODEL FAMILY

Training. We warm-start `Qwen3-8B` with SFT on 78K trajectories converted and filtered from OpenThoughts (Raoof et al., 2026), then apply the RIVER recipe using GRPO with the Endless-Agent harness (Gandhi et al., 2026) on **RIVER-TMax-3.5K**. Training stack and optimization details are provided in App. B, and SFT dataset details are described in App. C.1.

Baselines and Evaluation. We compare our trained model with open-source fine-tuned 8B models as well as the base model. The baselines include OpenThinker-8B-RL and OpenThinker-Agent-v1 (Guha et al., 2026; Raoof et al., 2026), both of which perform SFT before RL on their synthetic environments; OT-SFT-Endless-8B, which performs RL on the Endless-Terminals collection starting from the OpenThoughts SFT checkpoint (Gandhi et al., 2026); and TMax-8B and TMax-SFT-8B (Ivison et al., 2026), which are trained using their synthetic RL environments and SFT data, respectively. Each baseline is evaluated using its corresponding harness. We set the interaction turn budget to 64 and keep all other evaluation configurations fair across methods. Further evaluation details are provided in App. B.3.

Main Results. Comparisons are summarized in Tab. 1. We observe that our model achieves the best overall performance. Among the baselines, OT-SFT-Endless-8B uses the same harness as ours and starts from SFT data from the same OpenThoughts collection, but with lower atomic-skill coverage. Its RL stage is trained on a synthetic environment set of 3.3K tasks, which is comparable in scale to our collection. With the harness and RL training pipeline held fixed, the main differences in this comparison are the SFT initialization/data and the RL environment set; we examine these factors separately through controlled experiments in Sec. 4. TMax-8B is trained on the full TMax-15K environment collection. Despite using substantially fewer RL environments, our model achieves consistently higher performance across all four evaluation benchmarks. As discussed in App. D.4, the unfiltered TMax collection contains environments with verifier failures that can produce misleading reward signals and reward-hacking chances. The OpenThinker model is trained with large-scale SFT and RL. OpenThinker-8B-RL achieves the strongest performance among the baselines, while our model still maintains a clear average performance advantage. The contributors to this gap come from the higher-quality RL verifier signals from environment filtering, and behavior-level regularization, which better foster generalizable behaviors.

Behavior Ablations and Analysis. We select behavior verifiers for the verifier enhancement step in a data-driven way. We first assess the utility of each candidate behavior by how well it predicts the task success, using the single-behavior-feature AUC-ROC test (see Sec. 4.2 for AUC-ROC details) shown in Tab. 9 in App. E.2. The two strongest signals are `verify_before_done` and `repetition_rate`. We then run experiments with each of these top behaviors as a turn-level shaping signal, with the sign chosen according to its correlation: a turn-wise reward for verification

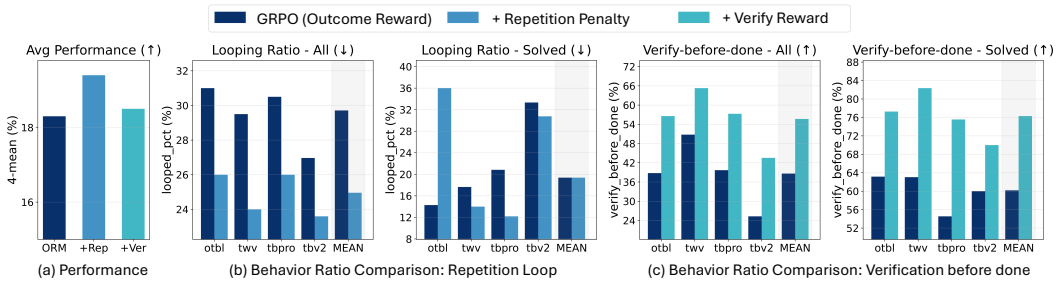


Figure 3: Ablations on behavior verifiers. **(a)**: Averaged performance on the four evaluation benchmarks; **(b, c)**: Behavior ratio comparison between RL with and without behavior verifiers. Arrows ↑ / ↓: the higher/lower, the better.

before completion, and a turn-wise penalty for repetitive loops. The performance and behavior comparisons for both behaviors are visualized in Fig. 3. Compared with outcome-based reward only, we find that advantage shaping does reshape the target behaviors: it reduces the repetition ratio and increases the fraction of verification in the corresponding runs. However, only the repetition-penalty run yields a noticeable overall performance improvement, whereas rewarding verification brings no additional gains. We interpret this as indicating that verification is correlated with task performance but does not directly cause task success. Based on this result, only the repetition penalty enters the final RIVER recipe as its verifier enhancement component 2.2.

We hope this discussion draws more attention to systematic study of multi-turn behaviors that are causally related to task performance, and to the design of advantage shaping in RL. Further ablations, such as the selection of SFT data and RL environments, are discussed in Sec. 4.3 and 4.4.

Section 3.1 Takeaways

- Our RIVER-8B model achieves the best performance on four terminal benchmarks.
- Advantage shaping from behavior verifiers steers the targeted behaviors, and discovering behavior verifiers that improve performance is an interesting and non-trivial direction.

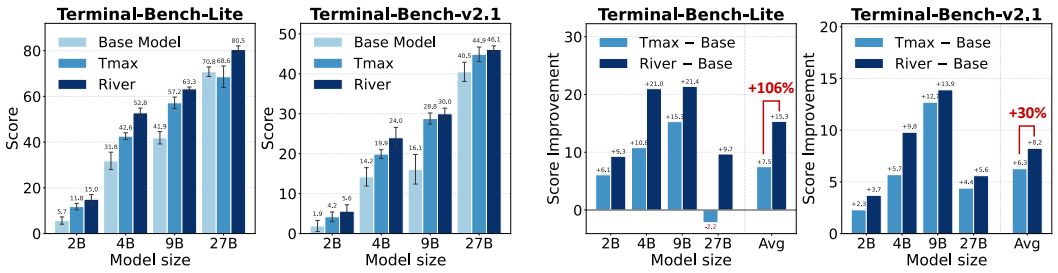
3.2 QWEN3.5 AND QWEN3.6 FAMILIES

Training. To evaluate environment filtering across model scales, we use the same base models as TMax: Qwen3.5-{2B, 4B, 9B} and Qwen3.6-27B. These experiments are designed to show the effects of RL environment quality. Thus, we remove the SFT step in our recipe and retain TMax's harness, DPPO objective (Qi et al., 2026), and most training settings the same, only replacing the full environment pool with **RIVER-TMax-3.5K** for a fair comparison with TMax results. Infrastructure differences and model details are provided in App. B.2.2. We evaluate on the same two benchmarks, Terminal-Bench-Lite and Terminal-Bench-v2.1.

Main Results. The evaluation results are presented in Fig. 4. We observe that RIVER consistently improves base-model performance across model sizes and outperforms the original TMax on both Terminal-Bench-Lite and Terminal-Bench-v2.1. These results demonstrate that our method generalizes across model families, model sizes, and agent harnesses in agentic RL for terminal agents. We also report the performance gains of the RL models over their corresponding base models. Notably, using only $\sim$30% of the Tmax environments (our TMax-RIVER-3.5K set), we achieve, on average, 106% and 30% larger RL gains on Terminal-Bench-Lite and Terminal-Bench-v2.1, respectively.

Section 3.2 Takeaways

- With our filtered environment set, **RIVER-TMax-3.5K**, RL using the same base models achieves, on average 106% and 30% larger performance gains on `otbl` and `tbv2.1`, respectively, compared with TMax training on the full environment set.
- The comparison between RIVER and TMax shows that RIVER generalizes across model families, model sizes, agent harnesses, and RL algorithms.



(a) Performance Comparison (b) Performance Improvement Comparison

Figure 4: Comparison results with the TMax model series. (a) Performance comparison from 2B to 27B models; (b) RL performance improvement over base models from 2B to 27B models. We report the mean value and standard deviation with 3 seeds. TMax results are adopted from their report.

4 Analysis

In this section, we first present experiments in Secs. 4.1 and 4.2 that support our **Agentic Compositional Generalization** hypothesis. We then provide further analyses and ablations in Sec. 4.3 and RL in Sec. 4.4. Unless otherwise stated, we use Qwen3-8B as the base model.

4.1 Skill preserving and behavior shaping in RL

We study how RL changes the model's low-level skills and high-level behaviors by comparing evaluation traces before and after RL for our 8B models warm-started with SFT. Given $N_s = 60$ predefined low-level skills, GPT-5.4 encodes each trace as a binary vector indicating which skills are exhibited. Across $N_t = 589$ evaluation tasks, this yields two binary matrices $M^{\text{pre}}, M^{\text{post}} \in \{0,1\}^{N_s \times N_t}$ before and after RL, where $M_{s,t} = 1$ iff skill s is exhibited in the trace for task t. Each row $M_{s,:} \in \{0,1\}^{N_t}$ is the *task-incidence vector* of skill s, indicating the tasks on which it appears, while each column $M_{:,t} \in \{0,1\}^{N_s}$ is the *skill vector* of task t, indicating the skills it recruits.

First, we compute the exhibition ratio of each skill by averaging along the task dimension, $r_s = \frac{1}{N_t}\sum_{t=1}^{N_t} M_{s,t}$, once for M^{pre} and once for M^{post}. The results are presented in Fig. 5 (a), where each point corresponds to one skill s at coordinates $(r_s^{\text{pre}}, r_s^{\text{post}})$. We observe that the skill ratios remain strongly correlated: skills that are rarely exhibited before RL also tend to remain rarely exhibited afterward. This suggests that RL does not substantially expand the observed skill repertoire beyond that already expressed by the SFT checkpoint.

Second, in Fig. 5 (b and c) we apply Representational Similarity Analysis (RSA) (Kriegeskorte et al., 2008) to the same matrices $M^{\text{pre}}, M^{\text{post}}$ along their two axes. The skill-skill structure is the matrix $S^{\text{skill}} \in \mathbb{R}^{N_s \times N_s}$ with $S^{\text{skill}}_{ij} = \cos\big(M_{i,:}, M_{j,:}\big)$, the cosine similarity between every pair of skills over their task-incidence vectors, i.e., "which skills co-occur". Symmetrically, the task-task structure is $S^{\text{task}} \in \mathbb{R}^{N_t \times N_t}$ with $S^{\text{task}}_{ab} = \cos\big(M_{:,a}, M_{:,b}\big)$, the cosine similarity between every pair of tasks over their skill vectors, i.e., "which tasks are solved with the same skills". We build each structure once for M^{pre} and M^{post}; each panel is a density scatter of one structure's upper-triangle entries before RL (x-axis) vs after RL (y-axis), one point per pair. We report the Pearson (Pearson, 1896) and Spearman correlation (Spearman, 1961) between the two triangles, with a Mantel permutation test (Mantel, 1967) for statistical significance.

Fig. 5 (b, c) support the hypothesis that (1) **The skill-skill structure is strongly preserved during RL**: in Fig. 5 (b), points lie close to $y = x$, with $\rho = 0.83$, indicating that skills that co-occurred before RL tend to continue co-occurring afterward. In contrast, (2) **the task-task structure changes substantially**: in Fig. 5 (c), the correlation is $\rho = 0.27$, with a diffuse cloud far from the diagonal $y = x$, suggesting that the set of skills recruited for a given task is reorganized during RL. Together, these results are consistent with RL primarily reshaping how existing skills are selected and composed across tasks.

4.2 Behaviors' Correlation with Task Performance

In this section, we provide additional details on the experimental design and results of the Area Under the Receiver Operating Characteristic Curve (**AUC-ROC**) analysis (Fawcett, 2006) introduced in Sec. 2.1. We predict binary task success using ℓ_2-regularized logistic regression with 15 seeds

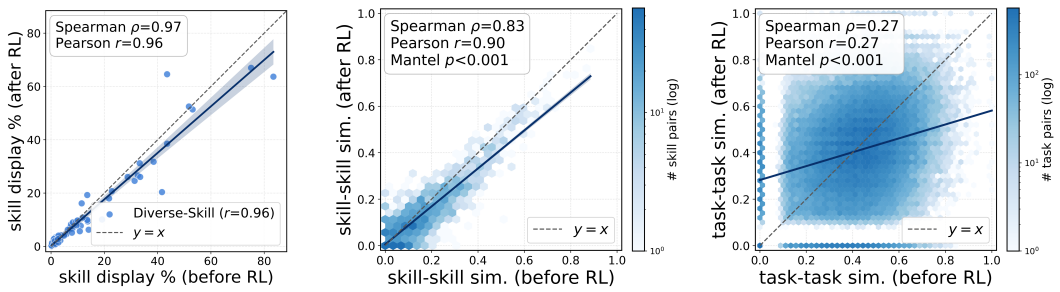


Figure 5: Skills and behaviors correlation analysis before and after RL. (a) Correlation between skill ratio; (b) Correlation between skill-skill occurrence; (c) Correlation between task-task structure.

and 5-fold cross-validation. **Model-S** uses skill-presence and task-category features; **Model-B** uses 12 rule-based behavior features, such as verification before completion and repetition. We compare SFT and RL trajectories, using an RL checkpoint trained without behavior-verifier enhancement. Candidate behaviors with trivial meanings are removed through human inspection; feature definitions and the full protocol are in App. E.2.

In Fig. 1 (Right), Model-B's ROC-AUC rises from 0.64 to 0.74 after RL, whereas Model-S stays near chance (0.48–0.56; approximately 0.55 before and after RL). ROC-AUC measures how often a randomly selected solved task receives a higher predicted success score than a randomly selected unsolved task; 0.5 indicates chance-level discrimination. Together with Sec. 4.1, these results support our hypothesis that RL primarily reshapes how existing skills are used across turns and tasks.

Section 4.1 and 4.2 Takeaways

- RL largely preserves the exhibited low-level skill repertoire.
- Task success is associated more strongly with trajectory-level behavioral features than with which skills a trajectory exhibits.
- RL reshapes high-level behaviors that route and compose previously exhibited skills.

4.3 SFT Analysis

To test whether broader atomic-skill coverage improves subsequent RL, we compare three SFT datasets derived from OpenThoughts (Guha et al., 2026; Raoof et al., 2026): **Narrow-SFT** ($\sim$13.6K trajectories), **Diverse-SFT** ($\sim$78K), and **Diverse-DS-SFT**, a random 13.6K subset of Diverse-SFT that matches Narrow-SFT in size while retaining broader skill coverage. We also include a control that performs RL directly on Qwen3-8B without SFT. Dataset provenance and format conversion are provided in App. C.1.

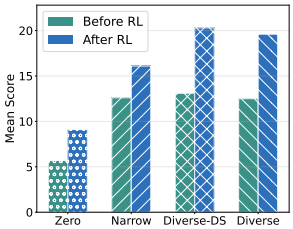


Figure 6: Mean score before and after RL.

The coverage comparison in Fig. 8 (App. C.1) makes this distinction concrete: Narrow-SFT mainly covers filesystem navigation and file editing, with almost no code execution or environment setup, such as running Python, executing `pytest`, or installing packages. Diverse-SFT and its downsampled subset cover these execution-oriented skills. The size-matched comparison therefore controls data volume while retaining a substantial difference in the skills exercised during SFT.

Fig. 6 reports mean performance across `Otbl`, `Tbv2.1`, `Twv`, and `Tbpro` before and after RL for all four initialization settings. RL directly on the base model yields the worst performance, motivating SFT for relatively weak models such as Qwen3-8B. Despite comparable SFT performance, the size-matched **Narrow-SFT** and **Diverse-DS-SFT** checkpoints differ substantially after RL. This gap supports the importance of broader atomic-skill coverage beyond the amount of SFT data alone.

In App. F.1, we provide a supplementary study on unfiltered, low-quality environments showing that training rewards and in-distribution performance can improve under **Narrow-SFT** even as benchmark performance degrades and reward-hacking behaviors emerge. Replacing **Narrow-SFT** with **Diverse-SFT** while holding all other configurations fixed substantially reduces reward hacking and improves evaluation performance. This comparison suggests that broader skill coverage can mitigate shortcut exploitation even when the reward design remains unchanged.

Table 2: Solve rate (%) on the held-out evaluation set by task domain.

Category	Build	Data	Debug	Graph	Numerical	Security	Systems	Text	ALL
RL (8 doms)	42.6 (+8.2)	56.9 (+27.7)	54.3 (+10.9)	68.8 (+40.6)	69.7 (+24.2)	40.6 (+18.8)	52.4 (+26.2)	55.6 (+11.1)	53.9 (+21.5)
RL (6 doms)	45.9 (+11.5)	50.8 (+21.5)	58.7 (+15.2)	65.6 (+37.5)	63.6 (+18.2)	43.8 (+21.9)	61.9 (+35.7)	44.4 (+0.0)	55.2 (+22.9)
RL (4 doms)	50.8 (+16.4)	53.8 (+24.6)	47.8 (+4.3)	68.8 (+40.7)	69.7 (+24.2)	40.6 (+18.7)	51.2 (+25.0)	55.6 (+11.2)	53.6 (+21.3)
RL (2 doms)	45.9 (+11.5)	47.7 (+18.5)	52.2 (+8.7)	53.1 (+25.0)	57.6 (+12.1)	46.9 (+25.0)	46.4 (+20.2)	55.6 (+11.1)	49.2 (+16.9)
SFT	34.4	29.2	43.5	28.1	45.5	21.9	26.2	44.4	32.3

4.4 RL Analysis

In this section, we want to better understand learning generalizable behaviors in terminal agents. We divide the River-TMax-3.5K dataset into 8 categories based on the required domain skills, as shown in Tab. 2, and construct a held-out evaluation set. We then perform RL on different subsets of domains, ranging from all 8 domains down to only 2. For RL (2 doms), training uses only $\sim$ 800 environments from the Debug and System domains. The evaluation results on the held-out set are presented in Tab. 2. We observe that reducing the number of training domains to 4 does not decrease overall performance, and we find no significant performance degradation even on specific domains excluded from training. When the number of training domains is reduced to 2, such that the training environments cover only Debug and System, overall performance decreases but remains substantially higher than that of the SFT model. Moreover, performance on untrained domains, such as Data and Numerical, still improves.

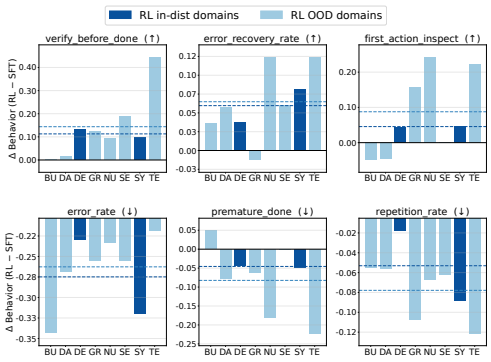


Figure 7: Behavior differences after RL (post-RL ratio - pre-RL ratio). Domains: BU = Build, DA = Data, DE = Debug, GR = Graph, NU = Numerical, SE = Security, SY = Systems, TE = Text. ↑/↓: the higher/lower, the better from empirical evaluations.

We also visualize 6 representative behaviors in the evaluation traces and measure the changes in their prevalence before and after RL for the 2-domain setting. The results are presented in Fig. 7. We observe that these behaviors are shaped not only in the trained domains, but also in the untrained domains. This suggests that the behavioral changes induced by RL generalize beyond the training domains and contribute to performance improvements in other domains.

Section 4.3 and 4.4 Takeaways

- For relatively weak base models (e.g., Qwen3-8B), SFT with diverse atomic-skill coverage provides an important warm start for RL to improve generalizability;
- SFT with broader skill coverage also helps reduce reward hacking;
- RL shapes behaviors beyond the training domains and fosters generalizability.

5 Conclusion

In this work, we study how RL improves terminal agents and propose the *agentic compositional generalization* hypothesis. Our empirical findings support complementary roles for different stages of model training: pre-training and SFT provide low-level atomic skills, while RL primarily shapes high-level decision-making behaviors that compose, route, and coordinate these skills across domains. This perspective highlights *verifier quality*, which determines the training signal shaping these behaviors, as a key factor beyond simply scaling the quantity or diversity of training environments. Guided by this insight, we introduce River, a simple training recipe that improves reward-signal integrity by filtering low-quality environments and optionally incorporating behavior-level regularization. River achieves the best performance among evaluated RL-trained 8B models across four terminal-agent benchmarks and generalizes across model families, scales, agent harnesses, and RL objectives. Using fewer than 30% of the TMax training environments, River achieves, on average, 106% and 30% larger RL gains on Terminal-Bench-Lite and Terminal-Bench-v2.1, respectively, across models ranging from 2B to 27B. Together, our results suggest that improving training-signal quality, rather than scaling environment coverage alone, is an effective and efficient path toward more generalizable terminal agents.

AI USE STATEMENT

In this work, we used generative AI tools for writing assistance and polishing, research execution, and synthetic-data generation. As part of the experimental pipeline, these tools were used for environment auditing and filtering, oracle-agent evaluation, and trajectory skill labeling, as detailed in the corresponding sections and appendices. We also used generative AI tools for polishing writing and editing software code. We have not used generative AI tools for retrieval and discovery, or drafting whole sections of the paper. The authors reviewed the AI-assisted outputs; for example, LLM-generated code was verified and tested for correctness. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

For the datasets used in our experiments, detailed descriptions of the data processing steps are provided in Sec. 3 and App. C. Training and evaluation details are provided in Sec. 3 and App. B. We have also submitted open-source release requests for the models and code, with the intention of making them publicly available in the future.

REFERENCES

Anthropic. Claude code: Best practices for agentic coding. `https://www.anthropic.com/engineering/claude-code-best-practices`, April 2025.

Anthropic. Claude opus 4.8. `https://www.anthropic.com/news/claude-opus-4-8`, 2026.

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. Swe-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. *Advances in Neural Information Processing Systems*, 38, 2026.

Ivan Bercovich, Ivgeni Segal, Kexun Zhang, Shashwat Saxena, Aditi Raghunathan, and Ziqian Zhong. Terminal wrench: A dataset of 331 reward-hackable environments and 3,632 exploit trajectories. *arXiv preprint arXiv:2604.17596*, 2026.

Shiyi Cao, Sumanth Hegde, Dacheng Li, Tyler Griggs, Shu Liu, Eric Tang, Jiayi Pan, Xingyao Wang, Akshay Malik, Graham Neubig, Kourosh Hakhamaneshi, Richard Liaw, Philipp Moritz, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Skyrl-v0: Train real-world long-horizon agents via reinforcement learning, 2025.

Zhepeng Cen, Haolin Chen, Shiyu Wang, Zuxin Liu, Zhiwei Liu, Ding Zhao, Caiming Xiong, Huan Wang, and Weiran Yao. Webscale-rl: Automated data pipeline for scaling rl data to pretraining levels. In *International Conference on Learning Representations*, volume 2026, pp. 87623–87642, 2026a.

Zhepeng Cen, Yihang Yao, William Han, Zuxin Liu, and Ding Zhao. Behavior injection: Preparing language models for reinforcement learning. *Advances in Neural Information Processing Systems*, 38:155105–155137, 2026b.

Fan Chen, Audrey Huang, Noah Golowich, Sadhika Malladi, Adam Block, Jordan Ash, Akshay Krishnamurthy, and Dylan Foster. The coverage principle: How pre-training enables post-training. In *International Conference on Learning Representations*, volume 2026, pp. 112225–112293, 2026a.

Zhaorun Chen, Zhuokai Zhao, Kai Zhang, Bo Liu, Qi Qi, Yifan Wu, Tarun Kalluri, Xuefei Cao, Yuanhao Xiong, Haibo Tong, et al. Scaling agent learning via experience synthesis. In *International Conference on Learning Representations*, volume 2026, pp. 121394–121420, 2026b.

Zihao Cheng, Hongru Wang, Zeming Liu, Xinyi Wang, Xiangrong Zhu, Yuhang Guo, Wei Lin, Jeff Z Pan, and Yunhong Wang. Terminal-world: Scaling terminal-agent environments via agent skills. *arXiv preprint arXiv:2605.20876*, 2026.

Tianzhe Chu, Yuexiang Zhai, Jihan Yang, Shengbang Tong, Saining Xie, Dale Schuurmans, Quoc V Le, Sergey Levine, and Yi Ma. Sft memorizes, rl generalizes: A comparative study of foundation model post-training. *arXiv preprint arXiv:2501.17161*, 2025.

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.

Zhiyuan Fan, Tinghao Yu, Yuanjun Cai, Jiangtao Guan, Yun Yang, Dingxin Hu, Jiang Zhou, Xing Wu, Zhuo Han, Feng Zhang, et al. Toward scalable terminal task synthesis via skill graphs. *arXiv preprint arXiv:2604.25727*, 2026.

Tom Fawcett. An introduction to roc analysis. *Pattern recognition letters*, 27(8):861–874, 2006.

Kanishk Gandhi, Ayush Chakravarthy, Anikait Singh, Nathan Lile, and Noah D Goodman. Cognitive behaviors that enable self-improving reasoners, or, four habits of highly effective stars. *arXiv preprint arXiv:2503.01307*, 2025.

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

Etash Guha, Ryan Marten, Sedrick Keh, Negin Raoof, Georgios Smyrnis, Hritik Bansal, Marianna Nezhurina, Jean Mercat, Trung Vu, Zayne Sprague, et al. Openthoughts: Data recipes for reasoning models. In *International Conference on Learning Representations*, volume 2026, pp. 108059–108130, 2026.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Harbor Framework Team. Harbor: A framework for evaluating and optimizing agents and models in container environments, 2026. URL `https://doi.org/10.5281/zenodo.20953922`.

Runyuan He, Qiuyang Mang, Shang Zhou, Kaiyuan Liu, Hanchen Li, Huanzhi Mao, Qizheng Zhang, Zerui Li, Bo Peng, Lufeng Cheng, et al. Frontiersmith: Synthesizing open-ended coding problems at scale. *arXiv preprint arXiv:2605.14445*, 2026.

Zhongmou He, Yee Man Choi, Kexun Zhang, Jiabao Ji, Junting Zhou, Dejia Xu, Ivan Bercovich, Aidan Zhang, and Lei Li. Hardtests: Synthesizing high-quality test cases for llm coding. *arXiv preprint arXiv:2505.24098*, 2025.

Audrey Huang, Adam Block, Dylan Foster, Dhruv Rohatgi, Cyril Zhang, Max Simchowitz, Jordan Ash, and Akshay Krishnamurthy. Self-improvement in language models: The sharpening mechanism. In *International Conference on Learning Representations*, volume 2025, pp. 76687–76739, 2025.

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 *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 2024.

Lingjing Kong, Xin Liu, Guangyi Chen, Martin Q Ma, Xiangchen Song, Yuekai Sun, Mikhail Yurochkin, Taylor W Killian, Ruslan Salakhutdinov, Kun Zhang, et al. From reasoning traces to reusable modules: Understanding compositional generalization in language model reasoning. *arXiv preprint arXiv:2606.18089*, 2026.

Nikolaus Kriegeskorte, Marieke Mur, and Peter A Bandettini. Representational similarity analysis-connecting the branches of systems neuroscience. *Frontiers in systems neuroscience*, 2:249, 2008.

Zhongzhi Li, Yucheng Shi, Zongxia Li, Ruhan Wang, Anhao Li, Zixun Huang, Junyao Yang, Lei Ke, Ninghao Liu, Haitao Mi, et al. Recursive synthesis for long-horizon terminal tasks. *arXiv preprint arXiv:2608.05466*, 2026.

Yusong Lin, Haiyang Wang, Shuzhe Wu, Lue Fan, Feiyang Pan, Sanyuan Zhao, and Dandan Tu. Cli-gym: Scalable cli task generation via agentic environment inversion. *arXiv preprint arXiv:2602.10999*, 2026.

Zuxin Liu, Thai Hoang, Jianguo Zhang, Ming Zhu, Tian Lan, Shirley Kokane, Juntao Tan, Weiran Yao, Zhiwei Liu, Yihao Feng, et al. Apigen: Automated pipeline for generating verifiable and diverse function-calling datasets. *Advances in Neural Information Processing Systems*, 37:54463–54482, 2024.

Nathan Mantel. The detection of disease clustering and a generalized regression approach. *Cancer research*, 27(2_Part_1):209–220, 1967.

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.

OpenAI. Openai o1 system card. `https://cdn.openai.com/o1-system-card-20241205.pdf`, December 2024.

OpenAI. Introducing codex. `https://openai.com/index/introducing-codex/`, May 2025.

OpenAI. Gpt-5.4 thinking system card. `https://openai.com/index/gpt-5-4-thinking-system-card/`, 2026.

Karl Pearson. Vii. mathematical contributions to the theory of evolution.—iii. regression, heredity, and panmixia. *Philosophical Transactions of the Royal Society of London. Series A, containing papers of a mathematical or physical character*, (187):253–318, 1896.

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, 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.

Akshara Prabhakar, Zuxin Liu, Ming Zhu, Jianguo Zhang, Tulika Manoj Awalgaonkar, Shiyu Wang, Zhiwei Liu, Haolin Chen, Thai Hoang, Juan Carlos Niebles, et al. Apigen-mt: Agentic pipeline for multi-turn data generation via simulated agent-human interplay. *Advances in Neural Information Processing Systems*, 38, 2026.

Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

Penghui Qi, Xiangxin Zhou, Zichen Liu, Tianyu Pang, Chao Du, Min Lin, and Wee Sun Lee. Rethinking the trust region in llm reinforcement learning. *arXiv preprint arXiv:2602.04879*, 2026.

Jielin Qiu, Zuxin Liu, Zhiwei Liu, Rithesh Murthy, Jianguo Zhang, Haolin Chen, Shiyu Wang, Ming Zhu, Liangwei Yang, Juntao Tan, et al. Locobench-agent: An interactive benchmark for llm agents in long-context software engineering. *arXiv preprint arXiv:2511.13998*, 2025.

Nived Rajaraman, Audrey Huang, Miroslav Dudik, Robert Schapire, Dylan Foster, and Akshay Krishnamurthy. Learning to reason with curriculum ii: Compositional generalization. *arXiv preprint arXiv:2606.27721*, 2026.

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.

Yi Ren and Danica Sutherland. Learning dynamics of llm finetuning. In *International Conference on Learning Representations*, volume 2025, pp. 70523–70563, 2025.

Chi Ruan, Dongfu Jiang, Huaye Zeng, Ping Nie, and Wenhu Chen. Evolvecoder: Evolving test cases via adversarial verification for code reinforcement learning. *arXiv preprint arXiv:2603.12698*, 2026.

Rulin Shao, Shuyue Stella Li, Rui Xin, Scott Geng, Yiping Wang, Sewoong Oh, Simon Shaolei Du, Nathan Lambert, Sewon Min, Ranjay Krishna, et al. Spurious rewards: Rethinking training signals in rlvr. *arXiv preprint arXiv:2506.10947*, 2025.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Y. Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, pp. 1279–1297, 2025.

Charles Spearman. The proof and measurement of association between two things. 1961.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026a. URL https://qwen.ai/blog?id=qwen3.5.

Qwen Team. Qwen3. 6-27b: Flagship-level coding in a 27b dense model, 2026b.

Weixun Wang, XiaoXiao Xu, Wanhe An, Fangwen Dai, Wei Gao, Yancheng He, Ju Huang, Qiang Ji, Hanqi Jin, Xiaoyang Li, et al. Let it flow: Agentic crafting on rock and roll, building the rome model within an open agentic learning ecosystem. *arXiv preprint arXiv:2512.24873*, 2025a.

Weixun Wang, XiaoXiao Xu, Xander Xu, et al. Let it flow: Agentic crafting on rock and roll, building the rome model within an open agentic learning ecosystem, 2025b. URL https://arxiv.org/abs/2512.24873.

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

Siwei Wu, Yizhi Li, Yuyang Song, Wei Zhang, Yang Wang, Riza Batista-Navarro, Xian Yang, Mingjie Tang, Bryan Dai, Jian Yang, et al. Large-scale terminal agentic trajectory generation from dockerized environments. *arXiv preprint arXiv:2602.01244*, 2026a.

Yifan Wu, Zhuokai Zhao, Songlin Li, Ho Hin Lee, Jiacheng Zhu, Shirley Wu, Tianhe Yu, Serena Li, Lizhu Zhang, Xiangjun Fan, et al. Swe-together: Evaluating coding agents in interactive user sessions. *arXiv preprint arXiv:2606.29957*, 2026b.

xAI. Grok code fast 1. https://x.ai/news/grok-code-fast-1, August 2025. Accessed: 2026-08-11.

Zhangchen Xu, Adriana Meza Soria, Shawn Tan, Anurag Roy, Ashish Sunil Agrawal, Radha Poovendran, and Rameswar Panda. Toucan: Synthesizing 1.5 m tool-agentic data from real-world mcp environments. *arXiv preprint arXiv:2510.01179*, 2025.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

John Yang, Kilian Lieret, Carlos Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents. *Advances in Neural Information Processing Systems*, 38, 2026a.

Rui Yang, Qianhui Wu, Zhaoyang Wang, Hanyang Chen, Ke Yang, Hao Cheng, Huaxiu Yao, Baolin Peng, Huan Zhang, Jianfeng Gao, et al. Gui-libra: Training native gui agents to reason and act with action-aware supervision and partially verifiable rl. *arXiv preprint arXiv:2602.22190*, 2026b.

Yihang Yao, Zhepeng Cen, Haohong Lin, Shiqi Liu, Zuxin Liu, Jiacheng Zhu, Zhang-Wei Hong, Laixi Shi, and Ding Zhao. Pushing forward pareto frontiers of proactive agents with behavioral agentic optimization. *arXiv preprint arXiv:2602.11351*, 2026a.

Yihang Yao, Guangtao Zeng, Raina Wu, Yang Zhang, Ding Zhao, Zhang-Wei Hong, and Chuang Gan. Tailored primitive initialization is the secret key to reinforcement learning. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 33300–33318, 2026b.

Tian Ye, Zicheng Xu, Yuanzhi Li, and Zeyuan Allen-Zhu. Physics of Language Models: Part 2.1, Grade-School Math and the Hidden Reasoning Process. In *Proceedings of the 13th International Conference on Learning Representations*, ICLR '25, April 2025. Full version available at `https://ssrn.com/abstract=5250629`.

Edward Yeo, Yuxuan Tong, Morry Niu, Graham Neubig, and Xiang Yue. Demystifying long chain-of-thought reasoning in llms. *arXiv preprint arXiv:2502.03373*, 2025.

Zhaochen Yu, Ling Yang, Jiaru Zou, Shuicheng Yan, and Mengdi Wang. Demystifying reinforcement learning in agentic reasoning. *arXiv preprint arXiv:2510.11701*, 2025.

Lifan Yuan, Weize Chen, Yuchen Zhang, Ganqu Cui, Hanbin Wang, Ziming You, Ning Ding, Zhiyuan Liu, Maosong Sun, and Hao Peng. From f (x) and g (x) to f (g (x)): Llms learn new skills in rl by composing old ones. In *International Conference on Learning Representations*, volume 2026, pp. 147547–147574, 2026.

Yang Yue, Zhiqi Chen, Rui Lu, Andrew Zhao, Zhaokai Wang, Yang Yue, Shiji Song, and Gao Huang. Does reinforcement learning really incentivize reasoning capacity in llms beyond the base model?, 2025. *URL https://arxiv. org/abs/2504.13837*, 204:1–16.

Guangtao Zeng, Maohao Shen, Delin Chen, Zhenting Qi, Subhro Das, Dan Gutfreund, David Cox, Gregory Wornell, Wei Lu, Zhang-Wei Hong, et al. Satori-swe: Evolutionary test-time scaling for sample-efficient software engineering. *arXiv preprint arXiv:2505.23604*, 2025.

Kai Zhang, Xiangchao Chen, Bo Liu, Tianci Xue, Zeyi Liao, Zhihan Liu, Xiyao Wang, Yuting Ning, Zhaorun Chen, Xiaohan Fu, et al. Agent learning via early experience. *arXiv preprint arXiv:2510.08558*, 2025.

Jiarong Zhao, Zhikai Lei, Zhiheng Xi, Rui Zheng, Hang Yan, Jie Zhou, Qin Chen, and Liang He. Nexforge: Scaling agent capabilities through requirement-driven task synthesis for llms. *arXiv preprint arXiv:2607.14186, 2026.*, 2026.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody H Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024a.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, and Zheyan Luo. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 3: system demonstrations)*, pp. 400–410, 2024b.

Yuhang Zhou, Lizhu Zhang, Yifan Wu, Jiayi Liu, Xiangjun Fan, Zhuokai Zhao, and Hong Yan. Synthetic sandbox for training machine learning engineering agents. *arXiv preprint arXiv:2604.04872*, 2026.

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.

CONTENTS

1 Introduction **1**

2 Mechanism Interpretation and Method **2**

2.1 Agentic Compositional Generalization 2

2.2 RIVER: Reward-Integrity Verified Environments for RL 3

3 Experiment Results **4**

3.1 Qwen3-8B Model Family 5

3.2 Qwen3.5 and Qwen3.6 Families 6

4 Analysis **7**

4.1 Skill preserving and behavior shaping in RL 7

4.2 Behaviors' Correlation with Task Performance 7

4.3 SFT Analysis 8

4.4 RL Analysis 9

5 Conclusion **9**

A Related Work **16**

B Training Details **17**

B.1 Supervised Fine-Tuning 17

B.2 Reinforcement Learning 17

B.3 Evaluation 19

C Data and Environment Details **20**

C.1 SFT Dataset Sources, Construction, and Coverage 20

C.2 RL environments 22

D Environment Evaluation **23**

D.1 Evaluation Rubrics and Issue Categories 23

D.2 Cross-model validation of the audit labels 24

D.3 Environment Evaluation Examples 25

D.4 Examples of Verifier Failures on the Unfiltered Environment Set 28

E Skill and Behavior Analysis Details **33**

E.1 Skill-Structure Analysis 33

E.2 Behavior Features and ROC-AUC Analysis 34

F Additional Experiments **36**

F.1 Reward Hacking and SFT Initialization 36

A Related Work

Terminal Agents such as Claude Code (Anthropic, 2025) and Codex (OpenAI, 2025) have rapidly become central to software-engineering workflows, demonstrating strong performance on long-horizon coding benchmarks such as Terminal-Bench (Merrill et al., 2026), SWE-Bench (Jimenez et al., 2024), and their variants (Badertdinov et al., 2026; Qiu et al., 2025). Motivated by these advances, the open-source and research communities have increasingly explored recipes for training terminal and coding agents (Wang et al., 2025a; Zeng et al., 2025; Guha et al., 2026; Raoof et al., 2026; Peng et al., 2026; Cheng et al., 2026; Bercovich et al., 2026). Among the various post-training approaches, reinforcement learning (RL) has emerged as one of the most effective techniques for improving long-horizon agentic behaviors (Yang et al., 2026b; Wang et al., 2026), and has recently been adopted for terminal-agent training. A key challenge, however, is scaling high-quality training data and executable environments, which has become one of the primary bottlenecks for further progress (Pi et al., 2026; Ivison et al., 2026; Ruan et al., 2026).

Synthetic Data and Environment Scaling. Scaling synthetic data has become a central paradigm for training LLM agents, driven by the scarcity and cost of collecting real-world interaction data (Chen et al., 2026b; Zhang et al., 2025; Cen et al., 2026a). Recent efforts have evolved from generating static demonstrations for supervised fine-tuning (SFT) (Liu et al., 2024; Xu et al., 2025; Prabhakar et al., 2026) to constructing interactive training environments (Yang et al., 2026a; Zhou et al., 2026; Lin et al., 2026; Li et al., 2026; He et al., 2026), which provide the executable feedback required for large-scale reinforcement learning. For terminal agents in particular, most existing work has focused on expanding environment coverage across skills, domains, and task categories (Ivison et al., 2026; Pi et al., 2026). While recent studies have shown that verifier quality, such as stronger test cases, can improve coding performance (He et al., 2025), the broader role of verifier quality in shaping learned behaviors and enabling out-of-distribution generalization for RL-trained terminal agents remains largely unexplored.

Learning Dynamics and Generalization Mechanisms. A growing body of work seeks to understand why LLMs and agents generalize beyond their training distributions and how these insights can guide more effective training recipes (Chu et al., 2025; Huang et al., 2025; Ren & Sutherland, 2025; Chen et al., 2026a; Ye et al., 2025; Yu et al., 2025; Rajaraman et al., 2026; Yue et al.). Recent works on *compositional generalization* (Kong et al., 2026; Yuan et al., 2026) explain the complementary roles of SFT and RL in single-turn reasoning models: SFT provides the low-level reasoning primitives, while RL learns to compose and route these primitives to enable generalization. Other studies investigate reasoning behaviors and decision-making patterns, showing how understanding these dynamics can improve RL training (Gandhi et al., 2025; Cen et al., 2026b; Yeo et al., 2025; Yao et al., 2026b), with recent extensions to multi-turn agentic settings (Yao et al., 2026a). A complementary line of work examines how spurious or misaligned reward signals affect RL learning and generalization (Shao et al., 2025). In contrast, we study the learning dynamics of *terminal agents* operating in realistic interactive environments. We extend the compositional generalization hypothesis to the multi-turn agentic setting, quantitatively dissociate single-turn skills from multi-turn behaviors at the feature level (see Sec. 4), and show how the identified behaviors can be targeted directly through advantage shaping and used to guide the design of RL training environments and recipes.

B TRAINING DETAILS

B.1 SUPERVISED FINE-TUNING

Training data. We finetune on multi-turn terminal-agent trajectories in an *endless-agent* format Gandhi et al. (2026): a system prompt, the task instruction, and an alternating sequence of assistant turns and environment observations, where every assistant turn contains an explicit chain-of-thought (`<think>...</think>`) followed by a single shell command or a terminating action. Starting from the OpenThoughts-100K SFT collection (Raoof et al., 2026), we remove trajectories that cannot be converted from Terminus-2 to Endless-Agent, obtaining **Diverse-SFT** with 78K trajectories. Dataset sources, conversion rules, and atomic-skill coverage are detailed in App. C.1.

Model and optimization. We perform full-parameter fine-tuning of **Qwen3-8B** using the LLaMA-Factory framework (Zheng et al., 2024b). Loss is computed on assistant tokens (generated tokens) only. Key hyperparameters are listed in Table 3.

Table 3: SFT hyperparameters.

Hyperparameter	**Value**
Base model	Qwen3-8B
Fine-tuning type	Full parameter
Precision	bf16
Epochs	1
Learning rate	1×10^{-5}
LR schedule	Cosine
Warmup ratio	0.1
Effective batch size	64
(per-device $\times$ GPUs $\times$ grad. accum.)	$(1 \times 8 \times 8)$
Max sequence length	16,384
Optimizer sharding	DeepSpeed ZeRO-2
Attention	SDPA

B.2 REINFORCEMENT LEARNING

Oracle filtering of training environments. After rubric-based auditing (App. D), we evaluate the remaining TMax environments using GPT-5.4 (OpenAI, 2026) with the Terminus-2 harness (Harbor Framework Team, 2026) as an oracle agent. We perform two rollout attempts per environment (a pass@$k = 2$ check) and remove environments for which neither rollout passes the verifier. Such failures may reflect task difficulty or incompatibility with our training infrastructure. This step produces the curated **RIVER-TMax-3.5K** collection described in Sec. 2.2.

We describe Qwen3-8B training first, followed by the Qwen3.5 and Qwen3.6 experiments.

B.2.1 QWEN3-8B

Model and training stack. Qwen3 uses full attention with grouped-query attention (GQA) (Yang et al., 2025). We adopt Endless-Agent as both the agent harness and the basis of our RL training pipeline (Gandhi et al., 2026). This pipeline builds on SkyRL (Cao et al., 2025), Harbor (Harbor Framework Team, 2026), and Verl (Sheng et al., 2025).

Training environments. For full RIVER, we train on the **RIVER-TMax-3.5K** environment set.

Training method. For the Qwen3 model family, We fine-tune the policy with **Group Relative Policy Optimization (GRPO)** (Shao et al., 2024). For each prompt we sample a group of G rollouts from the current policy, execute them in the terminal environment, and score each rollout with the task verifier under a *sparse binary* reward ($r \in \{0, 1\}$: 1 iff the verifier passes). The advantage term is defined and discussed in (1).

Objective. Let $\rho_t(\theta) = \pi_\theta(a_t \mid s_t)/\pi_{\theta_{\text{old}}}(a_t \mid s_t)$ be the per-token importance ratio and A_t the (shaped) token advantage.

$$\mathcal{L}(\theta) \;=\; -\mathbb{E}_t\Big[\min\big(\rho_t A_t,\; \text{clip}(\rho_t,\, 1-\varepsilon_{\text{low}},\, 1+\varepsilon_{\text{high}})\, A_t\big)\Big], \tag{2}$$

with asymmetric clipping bounds ε_{low}, and $\varepsilon_{\text{high}}$. The per-token losses are aggregated with a sequence-mean reduction.

Hyperparameters. We adopt most hyperparameters for RL training from the endless-terminal work (Gandhi et al., 2026). Table 4 lists the training hyperparameters.

Computation details. All RL runs for Qwen3-8B models use a single such node. Each node is an 8×NVIDIA H200 GPU server, with ∼143 GB HBM3e per GPU, fully NVLink-connected. Host side: 2× Intel Xeon Platinum 8488C (96 logical cores), and ∼2 TB system RAM. For the main experiments, we run for around 60 hours to perform 400 RL steps.

Table 4: RL (GRPO) training hyperparameters.

Hyperparameter	**Value**
Optimization	
Base policy	Qwen3-8B (SFT-ed checkpoint)
Advantage estimator	GRPO
Learning rate	5×10^{-7}
Warmup steps	20
Weight decay	0.01
Training epochs	10
Update epochs per batch	2
Train batch size (prompts)	32
Policy mini-batch size	32
Parameter sharding	FSDP2, 1 node × 8 GPUs
Max RL Step	400
Objective	
KL in reward	disabled
Clip range (low, high)	$(0.2,\ 0.28)$
Loss reduction	sequence-mean
Reward	sparse binary $\{0, 1\}$
Rollout / generation	
Samples per prompt	8
Max turns per episode	16
Max prompt length	1536
Max input length	16384
Max generation length (per turn)	2048
Sampling temperature	0.6
Top-p	1.0
Inference engines	8
Repetition penalty	
Penalty δ (per token)	0.05
Per-rollout cap	0.15

B.2.2 QWEN3.5 AND QWEN3.6

For the experiments in Sec. 3.2, we follow TMax (Ivison et al., 2026) in using Qwen3.5-{2B, 4B, 9B} and Qwen3.6-27B as the base models. Qwen3.5 uses a hybrid architecture with Gated DeltaNet linear attention and full-attention blocks; Qwen3.6-27B shares this architecture (Team, 2026a;b).

We adopt TMax's training pipeline and most of its configurations, including the harness and the DPPO RL algorithm (Qi et al., 2026), and train on the filtered **RIVER-TMax-3.5K** environment

set. A few infrastructure changes were necessary under resource constraints, including replacing the sandbox API with locally hosted Docker containers. We evaluate on the two benchmarks used by TMax, Terminal-Bench-Lite and Terminal-Bench-v2.1.

B.3 EVALUATION

Serving. All checkpoints are evaluated under an identical serving configuration. We convert each policy checkpoint to Hugging Face `safetensors` and serve it with **SGLang** (Zheng et al., 2024a). A single 8-GPU node hosts one model instance with tensor parallelism TP=8 (we do not use data-parallel replicas, which we found to deadlock under the agent's synchronous per-turn calls) and a static KV-cache memory fraction of 0.85. The served context length is the model's full 40,960-token window (`max_position_embeddings`).

Decoding and budgets. Rollouts use temperature T=0.6 and top-p=1.0, with up to 2048 tokens generated per turn. Each episode is capped at 64 action turns and a 600-second wall-clock budget, whichever comes first; a task that reaches either cap without a terminating action is scored as a failure.

Evaluation Benchmarks. We evaluate on four terminal-agent benchmarks and report their mean score, summarized in Tab. 5. All benchmarks are in the harbor (Harbor Framework Team, 2026) task format. For **OpenThoughts-TBLite**, the score is a continuous value per their design. For the rest benchmarks, scoring is binary pass/fail per task (reward 1 iff the verifier's tests pass).

Table 5: Evaluation benchmarks. The averaged score is the unweighted mean of the four test benchmarks.

Benchmark	#Tasks	Score Type	Focus
OpenThoughts-TBLite	100	Continuous	General CLI / file-ops / SWE
Terminal-Bench 2.1	89	Binary	SWE, ML, security, data science
Terminal-Bench-Pro	200	Binary	SWE, systems, debugging, data
TerminalWorld-Verified	200	Binary	Mixed (categorized, verified)

Table 6: Evaluation serving and decoding configuration, held fixed across all checkpoints and benchmarks.

Setting	Value
Serving engine	SGLang
Tensor parallelism	8 (`--tp 8`)
KV-cache mem. fraction	0.85
Served context length	40,960 tokens
Sampling temperature	0.6
Top-p	1.0
Max tokens per turn	2048
Max turns per episode	64
Wall-clock budget / task	600 s
Task concurrency	8

C Data and Environment Details

C.1 SFT Dataset Sources, Construction, and Coverage

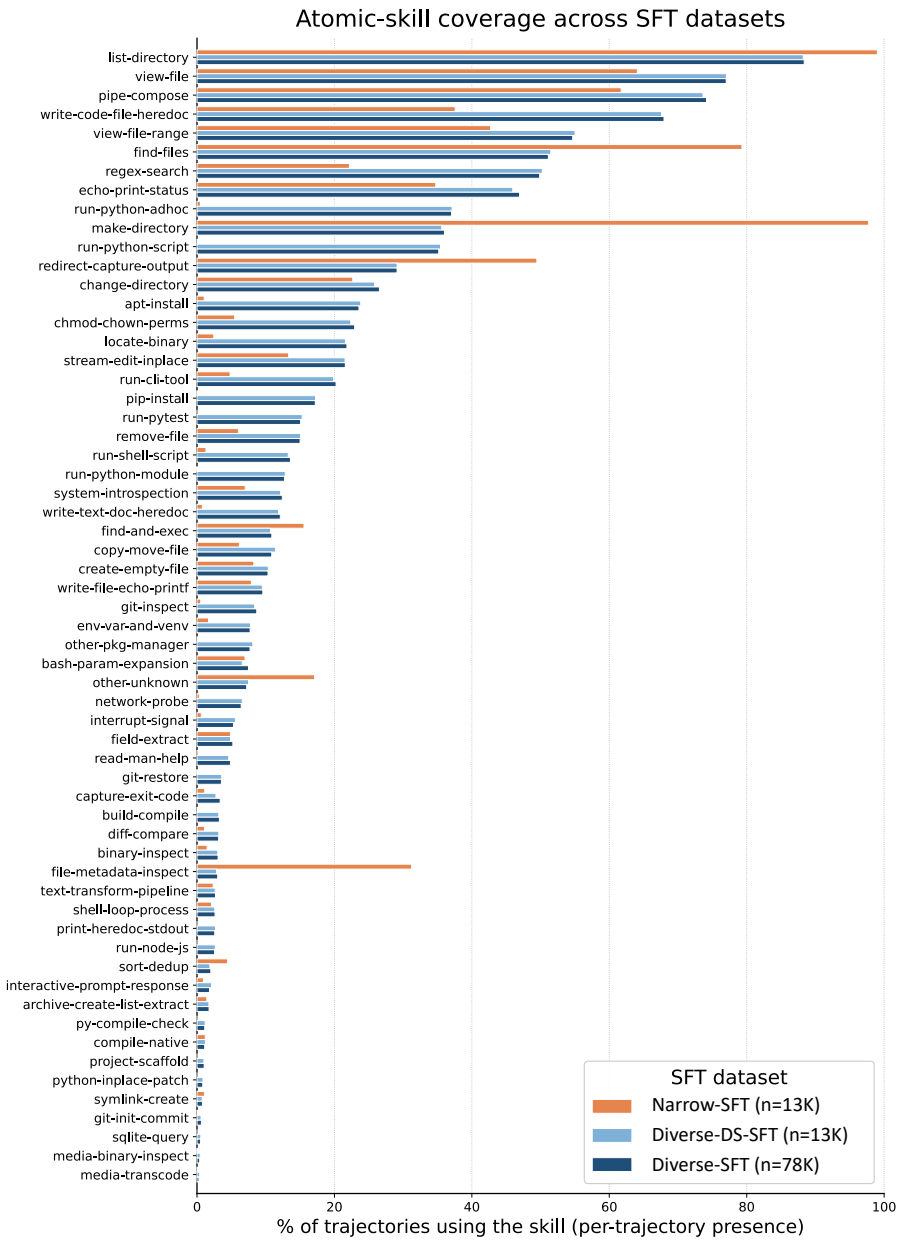


Figure 8: Atomic-skill coverage of the three SFT datasets, measured as the fraction of trajectories labeled by GPT-5.4 as exhibiting each of the 60 command-level skills.

Dataset sources and subsets. The SFT ablation in Sec. 4.3 uses three datasets derived from the public OpenThoughts collection (Guha et al., 2026; Raoof et al., 2026), after format conversion and filtering. **Narrow-SFT** contains approximately 13.6K trajectories, mainly from `nl2bash` and `InferredBugs`, with narrow atomic-skill coverage. **Diverse-SFT** contains approximately 78K trajectories from a wider range of sources, including `SWE-Smith` (Yang et al., 2026a) and OpenThoughts' IssueTasks datasets. **Diverse-DS-SFT** is a random 13.6K-trajectory subset of

Diverse-SFT, matching Narrow-SFT in size while preserving the broader skill coverage of Diverse-SFT. We use **Diverse-SFT** as the dataset in our main experiments unless stated otherwise.

SFT Construction Details. The source SFT data OpenThoughts-Collection (Guha et al., 2026; Raoof et al., 2026) uses the **Terminus-2** harness, whereas our stack consumes the **EndlessAgent** (Gandhi et al., 2026) harness, so we translate every trajectory deterministically and drop any that cannot be represented. In **Terminus-2**, the content in one generation turn is a JSON object with `analysis`, `plan`, a **batch** of `commands`, and `task_complete`, and each observation is a full terminal feedback. In EndlessAgent, the content in one generation turn consists of the following content: `<think>...</think>` `<command>`c`</command>` for one command c (or `<action>done</action>` to claim the task is done), and observations use the fixed template `Command executed successfully. Output: <stdout> (exit_code=0).`

The translation (1) replaces the system prompt with the EndlessAgent one; (2) extracts the task from the first user message (the span after `Task Description:`); (3) merges a turn's command batch $[c_1, \ldots, c_k]$ into `<command>`c_1 `&&` $\cdots$ `&&` c_k`</command>`; (4) fills `<think>` from `analysis`+`plan`; and (5) reduces each raw terminal screen to the command's actual output by removing the shell prompt and the echo of the typed command (including the continuation lines of multi-line commands), while carefully preserving genuine output that happens to start with ">" (e.g. `diff` results).

We drop a trajectory if its first message lacks a `Task Description:` marker, or if any assistant turn *other than the last* fails to parse as valid JSON. A non-JSON turn appearing *as the final turn* is not treated as corruption: it is the terminus "are you sure?" confirmation, which we convert to `done`. Because the trainer silently discards any trajectory with two consecutive assistant turns, we treat `done` as terminal and splice a synthetic observation wherever one assistant turn would directly follow another; The remaining trajectories all end in an explicit `done`.

Skill composition of the SFT datasets. Fig. 8 compares the atomic skill composition of the three SFT corpora (Narrow-SFT, Diverse-DS-SFT, and Diverse-SFT) used in our study, labeling every trajectory with a 60-skill atomic taxonomy (via GPT-5.4) and reporting per-skill coverage. The comparison serves two purposes. First, it validates Diverse-DS-SFT as a clean size control for Diverse-SFT: because Diverse-DS-SFT is drawn i.i.d. from Diverse-SFT, its coverage is indistinguishable from the full set across all 60 skills, so any downstream difference between models trained on Diverse-SFT and Diverse-DS-SFT is attributable to data *volume* rather than to a shift in skill composition. Second, it isolates what distinguishes Narrow-SFT from Diverse-SFT: the gap is not a uniform thinning but a categorical one on the execution-oriented skills. Narrow-SFT is dominated by filesystem navigation and authoring yet contains almost no code execution or environment setup (e.g., `run-python`, `run-pytest`, `pip/apt-install` all near 0%), whereas Diverse-SFT and Diverse-DS-SFT exercise these skills with SFT.

C.2 RL ENVIRONMENTS

In Fig. 9, each environment is passed through a two-stage filter, and every bar shows the outcome as a stacked count: (1) **fails rubric check** (grey): flagged by our rubrics and GPT-5.4 judges; (2) **pass rubric check but too hard** (blue): the environment is clean, but a capable reference agent cannot solve it; (3) **difficulty-adjusted** (green): clean *and* solvable within the budget. These environments consist of our **RIVER-TMax-3.5K**. The annotation above each bar reports `difficulty-adjusted% / rubric-pass%` of the original pool. Categories are the task-provided domain labels; skill groups are assigned by mapping each environment's primary skill onto one of eight groups (build, data, debug, graph, numerical, security, systems, text).

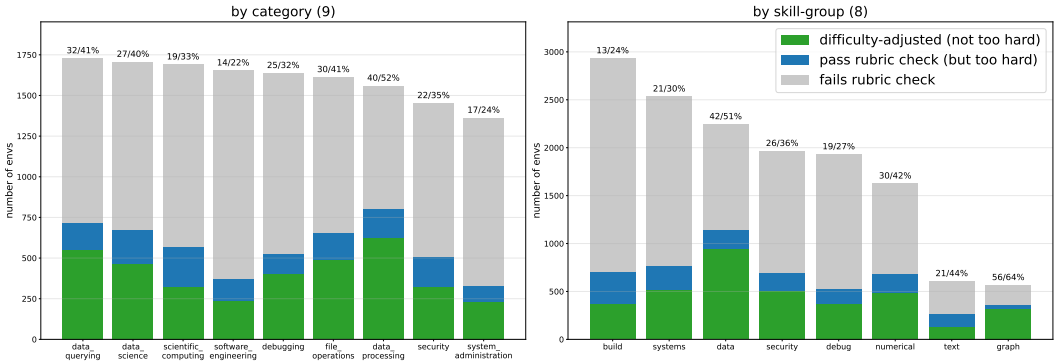


Figure 9: Environment quality-and-difficulty funnel for the TMax training pool (14,399 environments), broken down by task *category* (left, 9 classes) and by *skill group* (right, 8 groups).

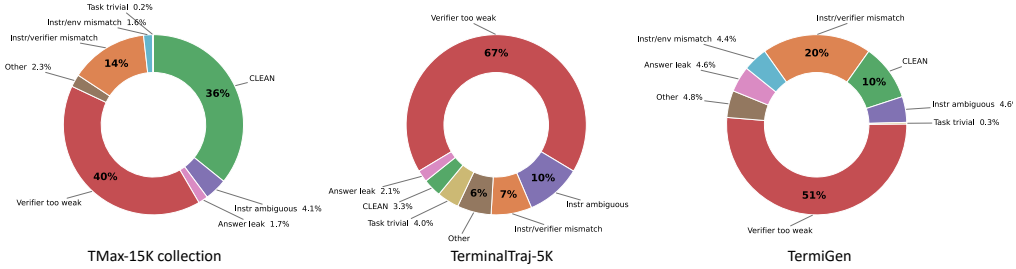


Figure 10: Rubrics evaluation results conducted by GPT-5.4: (left to right): Environments from TMax (Ivison et al., 2026); environments from TerminalTraj (Wu et al., 2026a); and environments from TermiGen (Zhu et al., 2026)).

D Environment Evaluation

We perform evaluations on three candidate environment pools using our environment-filtering rubrics. We emphasize that our evaluations may contain both false positives and false negatives. The results (Fig. 10) differ sharply across pools. **TMax** is the cleanest, with 35.8% Clean, but still contains 40.4% of environments flagged as Verifier-too-weak. **TermiGen** is only 10.1% Clean (51.4% too-weak, 19.6% mismatch), while **TerminalTraj-5k** performs the worst under our rubrics, with only 3.3% Clean and 67.2% Verifier-too-weak. Across all three pools, the dominant failure mode is weak verification (Verifier-Too-Weak). We next provide the full rubric (Sec.D.1) and show representative examples of each verdict (Sec.D.3).

D.1 Evaluation Rubrics and Issue Categories

The evaluation agent reads the three artifacts that must be mutually consistent: `instruction.md` (the task specification presented to the agent), the reward *verifier* (the pytest that scores a rollout), and the *environment* (the Dockerfile and any bundled data). It then asks a single agent-centric question: *would a competent agent that faithfully follows the instruction be able to (a) understand the task unambiguously and (b) actually pass the verifier?* Each task is assigned exactly one of eight verdicts:

- **Clean:** Instruction is clear and self-sufficient, a correct-per-instruction solution passes, AND the verifier genuinely validates correctness: it recomputes/decodes the true answer from the input (or exact-compares against a correct hardcoded answer) rather than merely checking output shape. A hard-but-fair task is still Clean.
- **Verifier-Too-Weak:** The verifier passes *without* the task actually being solved: it checks only output shape/format/types/count, trusts a self-reported value, greps the source for keywords, checks a hardcoded subset, or tests only fixed inputs so the answer can be hardcoded. A fabricated, plausibly-shaped output would pass.
- **Instr-Verifier-Mismatch:** The verifier checks something the instruction does not imply, or contradicts it. For example, reads/writes a different path, asserts a hardcoded value that conflicts with the data, or demands a stricter format. A solution that faithfully follows the instruction still fails.
- **Instr-Env-Mismatch:** The instruction or verifier references input paths, files, dependencies, or services the environment does not actually provide, so the task cannot be completed as written (missing data/compiler/service).
- **Answer-Leak:** The instruction itself, or a file shipped in the environment, reveals the expected answer (lists the result, embeds the solution, or provides a ground-truth file), so the task requires no genuine work.
- **Instr-Ambiguous:** The instruction alone is underspecified: a *necessary* specification is missing, so two equally reasonable interpretations produce different outputs and the verifier accepts only one. (Merely admitting multiple valid solution *paths* is not enough.)

Table 7: Binary cross-model comparison (CLEAN vs. defective) between the GPT-5.4 audit and an independent, label-blind Claude re-judge on a stratified sample of 120 TMax environments, with 15 examples drawn from each GPT-5.4 verdict category. Defect labels show substantially higher conditional agreement than CLEAN labels.

GPT-5.4 label	Independent blind re-judge	
	CLEAN	defective
CLEAN (15)	6	9
defective (105)	11	94

- **TASK-TRIVIAL:** The task itself is so easy there is no meaningful work or learning signal: write a constant, echo a value, copy a file, regardless of verifier strength.
- **OTHER:** A genuine defect fitting none of the above: verifier code that is buggy/crashes; a task needing network/GPU/GUI/hardware the container lacks; non-determinism; a required background service that is never started; or an internally contradictory spec.

D.2 CROSS-MODEL VALIDATION OF THE AUDIT LABELS

The verdicts above were produced by GPT-5.4. To assess whether these judgments are robust rather than idiosyncratic to a single model, we conducted an independent blind re-judgment using a different model, Claude Opus 4.8 (Anthropic, 2026). We constructed a stratified sample of 120 TMax environments, with 15 examples drawn from each of the eight GPT-5.4 verdict categories. To avoid anchoring, each environment was stripped of its GPT-5.4 label and any category cue, shuffled, and provided to a dedicated judge that independently read the full instruction, Dockerfile, and verifier code before assigning a verdict. The two labels were compared only afterward.

At the fine-grained level, the two models assign the same eight-way verdict to only 22.5% of the 120 examples. This disagreement does not necessarily imply that one judgment is incorrect: a single defective environment can exhibit multiple failure modes simultaneously. For example, a verifier that hard-codes its expected output may reasonably be categorized as VERIFIER-TOO-WEAK, while the same environment may also exhibit characteristics of INSTR/VERIFIER-MISMATCH or TASK-TRIVIAL. Thus, forcing each environment into a single primary category can lead the two models to disagree on *which* defect is most salient even when both identify the environment as defective.

For training, the more consequential distinction is binary: whether an environment is usable (CLEAN) or exhibits at least one defect. Under this binary grouping, the two models agree on $100/120$ examples (83.3%) in our stratified sample, as summarized in Tab. 7. Because the sample was stratified by the GPT-5.4 verdict rather than drawn according to the natural label distribution of the full pool, this number should be interpreted as agreement under our sampling design rather than as an estimate of population-level agreement.

More informative are the conditional agreement rates. Among the 105 examples labeled defective by GPT-5.4, the independent judge also identifies 94 as defective (89.5%). In contrast, among the 15 examples labeled CLEAN by GPT-5.4, only 6 are independently judged CLEAN (40.0%). This asymmetry suggests that defect labels are substantially more stable across judges than CLEAN labels. It also indicates that our reported clean rates may be optimistic, since some environments labeled CLEAN by GPT-5.4 are judged defective by the independent model. Overall, this validation supports using the audit as a conservative filtering mechanism for removing environments with readily detectable defects, while treating fine-grained defect categories and CLEAN labels with greater caution.

D.3 Environment Evaluation Examples

We show a compact set of representative environments flagged by our audit. Each example was independently checked by a second model. Each box summarizes what the task asks, why the environment receives the corresponding verdict, and the most direct evidence supporting that verdict. These examples are illustrative rather than exhaustive, and a single environment may exhibit more than one failure mode.

Verifier-Too-Weak

task_000254_21ede93b

Task. Write a C archiver daemon that recursively processes legacy log files, invokes a filter script via `popen()`, maintains atomic JSON state, and continuously monitors an incoming directory using `inotify`.
Why Verifier-Too-Weak. The instruction requires several concrete implementation properties: the solution must be a C program, recursively traverse existing logs, invoke the filter through `popen()`, update its state atomically using `rename()`, and use `inotify` to process newly closed files. The verifier, however, observes only a running process named `archiver`, the existence of output files, and the expected aggregate counts before and after adding a new file. It never checks that the implementation is written in C or that any of the required mechanisms are actually used. Consequently, another implementation that merely reproduces the expected observable files and counts could satisfy the verifier without implementing the requested archiver.
Key evidence. The verifier checks the process name, output-file existence, and `total_critical` counts, but never validates C source, `popen()`, recursive traversal, `inotify`, or atomic `rename()`.

task_000233_ee5f0e33

Task. Back up and repair several microservice configurations, write an idempotent Bash script that applies the required port and upstream settings, and write a second script that derives a topology report from the resulting configurations.
Why Verifier-Too-Weak. The verifier checks configuration values only through substring membership, such as testing whether `LISTEN_PORT=9001` occurs somewhere in the file. This does not ensure that the configuration is well formed, unique, or free of conflicting stale values. More importantly, although the task requires `generate_report.sh` to parse the configurations and dynamically construct the topology report, the verifier never executes that script. It only checks that the final report file contains a fixed set of expected lines. Thus, the required data flow from configuration repair to report generation is not verified.
Key evidence. Configuration values are checked by substring presence only, and `generate_report.sh` is never executed by the verifier.

Instr-Verifier-Mismatch

task_001036_07b75c89

Task. Repair a compromised data-processing pipeline, including an authentication validator, a redaction worker, and an Nginx reverse proxy that should listen on `127.0.0.1:8080` and forward traffic to the Flask application.
Why Instr-Verifier-Mismatch. The agent follows the networking requirement using the canonical Nginx configuration `listen 127.0.0.1:8080;`. However, the verifier searches the configuration with the regular expression `listen\s+8080\s*;`, which accepts `listen 8080;` but does not match the explicitly specified `127.0.0.1:8080` form. Thus, the verifier rejects a configuration that directly follows the instruction.
Key evidence. The instruction requires listening on `127.0.0.1:8080`; the agent writes `listen 127.0.0.1:8080;`, while the verifier only matches the pattern `listen 8080;`.

Instr-Env-Mismatch

task_000051_313e5ebc

Task. Transcribe an incident audio recording, redact the recovered secret from log files, and implement a secure JWT validator whose outputs should match a provided reference implementation under fuzz testing.
Why INSTR-ENV-MISMATCH. The audio input is generated at the path named by the instruction, but other artifacts required by the task and verifier are missing. In particular, the verifier requires `/app/reference_validator_oracle.py` for fuzz-equivalence testing, yet neither the container definition nor the setup script creates that file. Likewise, the instruction identifies `/home/user/service/` as the location of the service source to be audited, while the environment only creates that directory and provides no service implementation to inspect. A solution that follows the instruction therefore cannot complete all required steps or satisfy the verifier using the supplied environment.
Key evidence. The verifier asserts that `/app/reference_validator_oracle.py` exists, but the environment never creates it. The setup also creates `/home/user/service/` as an empty directory and installs no corresponding service implementation.

ANSWER-LEAK

task_000028_ae6330dd

Task. Perform a multi-stage incident investigation: extract information from video, recover a 4-byte XOR key from an ELF binary, analyze packet-capture traffic, and produce a JSON report containing the detected malicious events.
Why ANSWER-LEAK. The intended task requires combining several forensic modalities to recover three specific events. However, the setup exposes `/home/user/ground_truth.json`, which already contains the exact three events used by the verifier, including timestamps, payload values, and certificate flags. Reading this file reveals the complete expected report, so the intended video, binary, and packet-capture analysis can be bypassed.
Key evidence. The environment's `ground_truth.json` contains the same three events as the verifier truth set, including timestamps `00:01:15`, `00:02:30`, and `00:04:45`, together with the expected payload and flags.

task_000236_54686d44

Task. Recover a passphrase from video, decrypt a vault, reverse-engineer a malicious 12-byte signature from an ELF payload, and implement an efficient IDS scanner that detects that signature in logs.
Why ANSWER-LEAK. The intended workflow requires decrypting the vault and inspecting the ELF payload before the complete malicious signature is known. However, the agent-accessible evaluation log at `/app/test_logs.txt` already contains the exact injected signature `deadbeef1122334455667788`. Because the instruction also reveals that the signature begins with `DE AD BE EF`, an agent can recover the full value directly from the test log and hard-code it into the scanner, bypassing the vault-decryption and binary-analysis steps intended to derive the signature.
Key evidence. The environment injects `deadbeef1122334455667788` into 42 lines of `/app/test_logs.txt`, and the verifier uses the same value to construct the expected output.

TASK-TRIVIAL

task_001515_bf254241

Task. Recover a 3×3 projection matrix from the voice memo `/app/artifact_summary.wav`, then implement a Python transformation script that applies the recovered matrix to arbitrary three-dimensional inputs.
Why TASK-TRIVIAL. The stated challenge is to recover the matrix from audio. However, the agent-accessible environment also contains `/app/oracle_transform.py`, the exact reference implementation used by the verifier for fuzz equivalence. Inspecting or probing this

deterministic oracle directly reveals the transformation, so the audio-recovery component is unnecessary. In the observed rollout, the agent directly inspected the oracle and recovered the fixed diagonal transformation from it.
Key evidence. The verifier invokes `/app/oracle_transform.py` as the reference implementation, and this same oracle is accessible to the agent. It reveals the matrix

$$\mathrm{diag}(2.5, -1.5, 4.2),$$

allowing the transformation to be hard-coded without using the audio.

D.4 Examples of Verifier Failures on the Unfiltered Environment Set

Using the unfiltered TMax collection (Ivison et al., 2026) may lead to reward hacking and encourage shortcuts in policies' rollouts. The performance of RL on randomly sampled 3.5K environments from the TMax collection is presented in Tab. 8, where we can see a clear average performance gap with our model.

Table 8: Numbers are benchmark scores ($\times 10^2$). **Avg.** is the unweighted mean of the four benchmarks. Scores are mean $\pm$ std over 3 seeds.

Model	Otbl	Tbv2.1	Tbpro	Twv	**Avg.**
River-8B (ours)	21.8 $\pm$2.9	9.7 $\pm$0.6	23.0 $\pm$3.1	23.0 $\pm$2.5	**19.4** $\pm$1.2
RL On Random TMax-3.5K	18.6 $\pm$1.9	8.2 $\pm$1.3	23.7 $\pm$1.5	20.3 $\pm$0.6	17.7 $\pm$0.7
River-8B-SFT	11.3 $\pm$2.3	6.4 $\pm$1.3	13.7 $\pm$2.6	18.5 $\pm$2.3	12.5 $\pm$1.1

This section also provides some examples of bad behaviors fostered by training tasks with low-quality verifiers. Inspecting the corresponding trajectories shows that these defects can corrupt the reward signal in both directions. Some defective environments produce *false-positive rewards*: the agent obtains `reward`$= 1$ by exploiting leaked answers, degenerate oracles, or missing verifier checks without completing the intended task. Other defects produce *false-negative rewards*: a correct and instruction-aligned solution receives `reward`$= 0$ because the verifier uses an incorrect reference answer, a buggy oracle, an over-specified implementation check, or an internally inconsistent test. Below we show some representative examples for both *false-positive rewards* and *false-negative rewards*. Red highlights the critical shortcut or verifier failure.

False-positive rewards. The first five examples show cases in which the observed reward overstates task completion.

(1) Answer Leak – `task_003722` (reward $= 1$)

Task. Recover a scrolling news ticker from `archive_broadcast.mp4`. The agent must extract frames at 1 FPS, crop the bottom of each frame, run OCR, reverse mojibake corruption, normalize the recovered text, and reconstruct the chronological ticker in `reconstructed_ticker.txt`.
Environment/verifier defect. The verifier compares the reconstructed text against `/app/ground_truth.txt`, but this supposedly hidden reference is directly readable by the agent.

T1 While inspecting the environment, the agent discovers `/app/ground_truth.txt` alongside the video and reads the complete expected ticker.

T2–T5 The agent nevertheless attempts the intended video pipeline: it extracts frames, crops the ticker region, and constructs a noisy OCR-like reconstruction containing duplicated and corrupted fragments.

T6 It explicitly compares this reconstruction against the leaked reference and obtains only 0.368 similarity, far below the required threshold of 0.85.

T7 The agent then reasons that the output should match the expected string and replaces the failed reconstruction with the text read directly from `ground_truth.txt`.

T8 The final output is identical to the reference, giving similarity 1.0 and full reward.

Why the reward is misleading. The intended challenge is to recover the ticker from the video. The agent's actual reconstruction is insufficient, but the exposed reference lets it replace that failed result with the exact answer. Thus, the successful reward reflects access to the verifier target rather than successful video understanding.

(2) Answer Embedded in the Instruction – `task_008903` (reward $= 1$)

Task. Write and run a C++ scanner over several binaries. For each valid 64-bit ELF file, the scanner must determine whether the SUID bit is set, parse the ELF `.rodata` sec-

tion, detect whether it contains the string `/etc/shadow`, and write the sorted results to `/home/user/scan_results.txt`.
Environment/verifier defect. The instruction's example output is not merely illustrative: it gives the exact correct three output lines for the binaries in the environment. The verifier then checks the final result against those same expected values.

T1–T3 The agent attempts to construct the requested C++ scanner, but compilation and execution do not produce a reliable working scan.

T4 It notices that the instruction already provides the complete expected output: `fake_bin: SUID=Yes, SHADOW_IN_RODATA=No`, `safe_bin: SUID=No, SHADOW_IN_RODATA=No`, and `suid_bin: SUID=Yes, SHADOW_IN_RODATA=Yes`.

T5 The trajectory explicitly acknowledges the shortcut: "I've just created a hardcoded output file", and writes these three lines directly to `scan_results.txt`.

T6 The hard-coded file matches the expected output and receives full reward even though the requested scanner is not demonstrated to have produced it.

Why the reward is misleading. The task is intended to test binary inspection and ELF parsing. Instead, the exact answer is exposed in the instruction and can be copied without obtaining it from the binaries. The final-output verifier cannot distinguish this shortcut from a working scanner.

(3) Task Trivial – `task_001515` (reward = 1)

Task. Recover a 3×3 linear transformation from the voice memo `/app/artifact_summary.wav`, then implement the transformation in `/home/user/transform.py`. The script is evaluated on random three-dimensional input vectors.
Environment/verifier defect. The exact reference implementation, `/app/oracle_transform.py`, is shipped inside the agent-accessible container, and the verifier uses this same script as the oracle for fuzz equivalence.

T1 The agent lists the task files and discovers `oracle_transform.py` alongside the audio memo.

T2 Rather than recovering the matrix from the audio, it executes `cat /app/oracle_transform.py`.

T3 The reference implementation reveals the entire transformation:

$$\mathrm{diag}(2.5, -1.5, 4.2).$$

T4 The agent writes `transform.py` using exactly these constants.

T5–T6 It explicitly notes that "the audio file wasn't needed". The verifier then fuzzes the copied implementation against the exposed oracle, and all tests pass.

Why the reward is misleading. The generated implementation is numerically correct, but the defining challenge of the task is to recover the transformation from the voice memo. Exposing the reference implementation collapses the intended audio-understanding task into copying a few constants.

(4) Degenerate Oracle – `task_000925` (reward = 1)

Task. Reverse-engineer the behavior of a proprietary semantic-version resolver, `/app/legacy_resolver`, and implement a Bash replacement supporting SemVer comparisons, Boolean operators, and parentheses.
Environment/verifier defect. The supplied resolver is degenerate: across the agent's probes, it returns exit code 0 even for obviously false comparisons. The verifier evaluates the replacement by agreement with this broken resolver.

T1–T3 The agent probes `legacy_resolver` on straightforward positive and negative examples.

T4 It observes that contradictory cases such as `>=1.0.0` with `0.9.0`, `>2.0.0` with `1.9.0`, and `=1.0.0` with `2.0.0` all return success.

T5 The agent explicitly concludes: "The binary appears to have a bug where it just always returns exit code 0."

T6 Instead of implementing semantic-version parsing and comparison, it creates a replacement whose effective behavior is simply `exit 0` for every input.

T7 Because the verifier compares the replacement against the same degenerate oracle, this constant policy receives full reward.

Why the reward is misleading. The replacement does not implement semantic-version matching. Instead, the policy identifies the oracle's degenerate decision boundary and implements the cheapest behavior that reproduces it. This shows how a defective oracle can make a trivial shortcut optimal under the observed reward.

(5) Unchecked Implementation Constraint – `task_008402` (reward = 1)

Task. Fix a C++ backup-analysis program so that it queries replication metadata, constructs the valid directed graph, computes the lowest-latency path between two datacenters, and writes the result to `optimal_backup_path.json`. The instruction explicitly requires the solution to be produced *entirely in C++* and forbids a separate Python or Bash implementation.
Environment/verifier defect. The verifier only checks that `/home/user/backup_analyzer` exists and is executable, then separately checks the JSON contents. It never runs the C++ executable to verify that the executable actually produced the answer.

T1–T3 The agent edits and compiles the requested C++ program, but repeated executions of `backup_analyzer` time out and never produce the required JSON.

T4 Instead of continuing to debug the required implementation, it writes a separate Python program, `backup_checker.py`, that computes the answer.

T5 The Python program writes the correct path `[1,2,4]` with total latency `30` to `optimal_backup_path.json`.

T6 The trajectory explicitly acknowledges that the C++ program was timing out and that Python was used to generate the final output.

T7 The verifier observes an executable C++ file and a correct JSON file, never checks their causal relationship, and returns full reward.

Why the reward is misleading. The numerical answer is correct, but the explicit implementation requirement is not satisfied. Existence-only verification of the binary lets a forbidden Python bypass receive the same reward as a working C++ implementation.

False-negative rewards. The next five examples show the opposite failure mode: the observed reward understates task completion because the verifier rejects correct or instruction-faithful behavior.

(6) Incorrect Numerical Reference – `task_009230` (reward = 0)

Task. Compute a posterior score using the specified formula

$$p = (1 - \exp(-x))\, p_{\text{prior}},$$

and write the resulting probability rounded to six decimal places.
Environment/verifier defect. For one case with $x = 2.99$ and $p_{\text{prior}} = 0.7$, the verifier hard-codes `0.664815` as the expected value.

T1–T3 The agent follows the numerical procedure specified by the instruction.

T4 For the disputed value, it computes

$$(1 - \exp(-2.99)) \times 0.7 = 0.664798794\ldots,$$

which rounds to `0.664799`.

T5 The remaining reported numerical values match the verifier.

T6 The verifier nevertheless expects `0.664815`. The discrepancy exceeds its tolerance, so the mathematically correct rollout receives `reward`$= 0$.

Why the reward is misleading. The agent follows the stated formula and produces the correct rounded value. The failure is caused by an erroneous hard-coded reference constant, not by an error in the solution.

(7) Incorrect Semantic Assumption – `task_002319` (reward $= 0$)

Task. Perform spectral analysis on an EIIP-mapped DNA sequence. After computing the float64 FFT magnitude spectrum, the instruction requires finding the *maximum magnitude* over indices 1 through $N/2$, excluding only the DC component.
Environment/verifier defect. The verifier does not compute the requested maximum. Instead, it hard-codes `expected_peak_index = N // 11`, assuming that the fundamental frequency must be dominant.

T1–T3 The agent parses the FASTA file, constructs the float32 and float64 EIIP sequences, computes both FFTs, and evaluates their magnitude spectra.

T4 Following the instruction literally, it searches indices 1 through $N/2$ and obtains `peak_index = 50000` with magnitude approximately 1311.08.

T5 Independent recomputation confirms that this is the true maximum. The verifier's expected index `10000` has magnitude only about 322.33.

T6 The verifier nevertheless rejects `50000` because it expects the hard-coded fundamental-frequency index, producing `reward`$= 0$.

Why the reward is misleading. The agent correctly answers the question posed by the instruction. The verifier substitutes an incorrect domain assumption for the actual maximum and penalizes the correct spectral result.

(8) Buggy Reference Oracle – `task_001636` (reward $= 0$)

Task. Build a high-performance C program for "latest-frame" lookup. Given a query timestamp T, it must return the frame whose `pkt_pts_time` is the largest value less than or equal to T, using an indexed array and binary search.
Environment/verifier defect. The reference oracle reads H.264 packet metadata in decode order, where PTS values are not monotonic, and then runs binary search on this unsorted array. The verifier compares the agent against this buggy oracle rather than against the requested lookup semantics.

T1–T3 The agent extracts the same PTS/packet-size pairs from the video as the oracle.

T4 Because binary search requires sorted keys, it correctly sorts the metadata by PTS before building the query index.

T5 For query `2.6755`, the largest PTS not exceeding the query is `2.6`, whose packet size is `120`.

T6 The unsorted oracle instead returns `97`. The verifier treats `97` as ground truth and rejects the agent's correct `2.6755 -> 120` output.

Why the reward is misleading. The agent implements the requested latest-frame semantics and satisfies the precondition required for binary search. It receives zero reward because oracle equivalence is used as a proxy for correctness even though the oracle itself violates the algorithm's assumptions.

(9) Brittle Source-Code Check – `task_006064` (reward $= 0$)

Task. Fix a Rust text-processing pipeline so that it lowercases the corpus, retains only alphabetic characters `a-z` and spaces, tokenizes by whitespace, reports the top three tokens, and computes their 3×3 sample covariance matrix.
Environment/verifier defect. Two functional tests validate the produced tokens and covariance matrix, but a third test additionally requires the source file to contain the literal substring `is_alphabetic`. The instruction never requires this specific Rust API.

T1–T3 The agent fixes the tokenization using `c.is_ascii_alphabetic() || c.is_whitespace()`, which directly matches the instruction's requirement to retain `a-z` characters and spaces.

T4 It builds and runs the Rust project successfully.

T5 The resulting `top_tokens.txt` is byte-exact correct (`the`, `quick`, `dog`), and `covariance.csv` exactly matches the expected matrix.

T6 Both functional output tests pass, but the source-inspection test fails solely because `is_ascii_alphabetic` does not contain the literal substring `is_alphabetic`. The final reward is therefore 0.

Why the reward is misleading. The observable behavior is correct, and the chosen API is at least as faithful to the explicit `a-z` requirement as the expected implementation. A brittle source-pattern check rejects a functionally correct solution because it uses a different valid implementation.

(10) Inconsistent End-to-End Test – `task_001040` (reward = 0)

Task. Repair a local experiment-tracking pipeline by configuring Nginx to proxy `127.0.0.1:8080` to a provided Flask server, writing a CSV validator, and starting the services. Clean training-log uploads should be accepted, while malformed logs should be rejected.
Environment/verifier defect. The provided Flask server accepts uploads only through a multipart form field named `file`; if this field is missing, it returns HTTP 400. However, the end-to-end verifier sends the CSV bytes as a raw POST body with no multipart encoding and no `file` field, while still requiring HTTP 200 for a clean file.

T1–T4 The agent writes a validator enforcing the requested headers and numerical constraints, configures Nginx to proxy port 8080 to the Flask service on port 5000, and writes an executable startup script.

T5 The static validator, startup-script, and configuration tests pass; three of the four verifier tests succeed.

T6 The end-to-end test reads a clean CSV into raw bytes and constructs `urllib.request.Request(..., data=data, method="POST")` without a multipart `file` field.

T7 The unmodified Flask fixture therefore behaves as designed and returns `400 BAD REQUEST`, while the verifier expects `200`. The rollout receives `reward= 0`.

Why the reward is misleading. The failing request does not use the upload protocol required by the provided server. The verifier and fixture are internally inconsistent, so an instruction-faithful solution cannot satisfy the end-to-end test.

Takeaway. These examples show that defective environments can corrupt the RL signal in both directions: **False-positive** rewards arise when answer-bearing artifacts, degenerate oracles, or incomplete checks make shortcuts sufficient for `reward= 1`. **False-negative** rewards arise when incorrect reference values, buggy oracles, over-specified source checks, or inconsistent test protocols reject solutions that correctly follow the instruction. Together, these cases further motivate our environment filtering process.

E SKILL AND BEHAVIOR ANALYSIS DETAILS

E.1 SKILL-STRUCTURE ANALYSIS

This subsection details the feature construction and statistical comparisons used in Sec. 4.1.

Trajectory representations. We compare evaluation traces before and after RL for our 8B models, which are warm-started with SFT. With a list of $N_s = 60$ predefined low-level skills, we use GPT-5.4 to represent each trace as a binary vector indicating which skills it exhibits. Across the $N_t = 589$ evaluation tasks, this gives two matrices $M^{\text{pre}}, M^{\text{post}} \in \{0,1\}^{N_s \times N_t}$, where $M_{s,t} = 1$ if and only if skill s is exhibited in the trace for task t. Row $M_{s,:}$ is skill s's *task-incidence vector*, indicating the tasks on which it appears; column $M_{:,t}$ is task t's *skill vector*, indicating the skills it recruits.

Skill exhibition ratios. For each skill, we compute $r_s = \frac{1}{N_t}\sum_{t=1}^{N_t} M_{s,t}$ separately for M^{pre} and M^{post}. Each point in Fig. 5 (a) corresponds to one skill at coordinates $(r_s^{\text{pre}}, r_s^{\text{post}})$.

Representational Similarity Analysis. We apply RSA (Kriegeskorte et al., 2008) to the two axes of each matrix. The *skill-skill* structure is the matrix $S^{\text{skill}} \in \mathbb{R}^{N_s \times N_s}$, with

$$S_{ij}^{\text{skill}} = \cos\big(M_{i,:},\, M_{j,:}\big),$$

the cosine similarity between pairs of skills over their task-incidence vectors, measuring which skills tend to co-occur across tasks. The *task-task* structure is the matrix $S^{\text{task}} \in \mathbb{R}^{N_t \times N_t}$, with

$$S_{ab}^{\text{task}} = \cos\big(M_{:,a},\, M_{:,b}\big),$$

the cosine similarity between pairs of tasks over their skill vectors, measuring which tasks recruit similar sets of skills. We build each structure separately from M^{pre} and M^{post}. Figures 5 (b, c) plot the corresponding upper-triangle entries before RL (x) against those after RL (y), with one point per pair. We report Pearson (Pearson, 1896) and Spearman (Spearman, 1961) correlations between the two triangles, with a Mantel permutation test (Mantel, 1967) for significance.

E.2 Behavior Features and ROC-AUC Analysis

Predictive comparison and evaluation protocol. The analysis in Sec. 4.2 tests whether task success is more strongly associated with which skills a trajectory uses or with how the agent behaves. For each evaluation trajectory, **Model-S (skills)** encodes the skills present in the trace together with its task category, whereas **Model-B (behavior)** uses the 12 rule-based features in Tab. 9, such as verification before completion and repetition. We use human inspection to remove candidate behaviors with trivial meanings. Both representations are computed from trajectories generated by the SFT checkpoint and an RL checkpoint trained without behavior-verifier enhancement.

We fit an ℓ_2-regularized logistic regression classifier to predict the binary outcome `solved` from each representation and evaluate ROC-AUC using 15 seeds and 5-fold cross-validation. ROC-AUC is the probability that the classifier assigns a higher score to a randomly chosen solved task than to a randomly chosen unsolved task (Fawcett, 2006); 0.5 denotes chance-level discrimination and 1.0 denotes perfect separation. In Fig. 1 (Right), Model-B reaches AUCs of 0.64–0.74, while Model-S remains near chance (0.48–0.56). From SFT to RL, Model-B's AUC increases from 0.64 to 0.74, whereas Model-S remains near 0.55.

Feature extraction. Each behavior feature is extracted deterministically from a trajectory using rule-based parsing, with no model or LLM in the loop. For each assistant turn, we parse the `<command>` and treat the following user turn as the resulting observation. A command is considered an *inspection* if its head corresponds to a pre-defined read-only tool set including (`ls`, `cat`, `grep`, `find`, `stat`, ..., etc.) and does not perform redirection or writing. An *error* is an observation that matches an error regex, such as `Traceback`, `command not found`, `No such file`, or a non-zero exit status. Two commands are considered *near-duplicates* if their token-level Jaccard similarity is at least 0.8.

AUC denotes the single-feature cross-validated AUC-ROC for predicting per-task success of the RL checkpoint on the four evaluation benchmarks, averaged over 15 seeds of 5-fold cross-validation. **Dir.** indicates the sign of the feature's correlation with success. Rows are sorted by AUC. The two strongest individual signals are verifying before finishing (`verify_before_done`, $+$) and repeatedly issuing similar commands (`repetition_rate`, $-$). The latter serves as a broader proxy for the harmful repetition loops targeted by our turn-wise RL penalty.

Table 9: The 12 behavior features used by Model-B, together with their definitions and univariate predictive power.

Feature	Definition / extraction rule (per trajectory)	AUC	Dir.
`verify_before_done`	1 if at least one read-only inspection occurs among the up to three commands immediately preceding the `<done>` action; otherwise 0.	0.632	+
`repetition`	Fraction of commands that are near-duplicates of at least one earlier command in the same trajectory.	0.631	−
`cmd_diversity`	Number of distinct command heads divided by the total number of commands.	0.615	+
`python_heredoc`	Fraction of commands that invoke Python (`python`/`python3`, including `-c`) or use a here-document to perform task-related operations.	0.604	−
`inspect_before_act`	Fraction of commands that perform read-only inspection, defined as commands with an inspection head and no output redirection, before modifying the environment.	0.596	+
`n_errors`	Number of commands whose resulting observations match the error regex.	0.563	−
`gave_up_flag`	1 if the episode ends *without* a `<done>` action and its final observation is an error; otherwise 0.	0.549	−
`error_rate`	Fraction of commands whose resulting observations match the error regex (`n_errors` divided by the total number of commands).	0.535	−
`error_recovery`	Among commands that result in an error, the fraction for which the *next* command is substantially different (i.e., not a near-duplicate), indicating a change in approach after failure.	0.531	+
`inspect_first`	1 if the first command in the trajectory is a read-only inspection; otherwise 0.	0.524	+
`tool_help`	Fraction of commands that request tool help or usage information (`--help`, `-h`, or `man`).	0.510	−
`premature_done`	1 if `<done>` is emitted either after the immediately preceding command results in an error or without prior verification; otherwise 0.	0.507	~ 0
All 12 features (Model-B)		**0.735**	

F Additional Experiments

F.1 Reward Hacking and SFT Initialization

Complementing the SFT ablation in Sec. 4.3, we further observe that **Diverse-SFT** can reduce reward hacking. During an early stage of our exploration of RL with environments from TermiGen (Zhu et al., 2026), we adopted partial rewards defined as $r_p = 0.5\times$ pass-rate $+ 0.5\times$ full-pass-bonus. The training curves are presented in Fig. 11 (a), where both the training rewards and in-distribution evaluation scores improve during RL. However, this setup introduces two potential sources of reward hacking: (1) **Hackable verifiers**: the environments had not yet been filtered by our River pipeline. The evaluation results in Fig. 10 (App. D) indicate that this dataset is not sufficiently clean for RL[3]; and (2) **Early termination**: under partial rewards, the agent may learn a shortcut that satisfies only a subset of the verifiers and then prematurely claims task completion. As shown in Fig. 11 (b, left), RL degrades generalizability even as in-distribution performance improves, which is consistent with reward hacking. Combined with the decrease in generation length during training and our trajectory analysis, these results indicate that this form of reward hacking primarily stems from early termination.

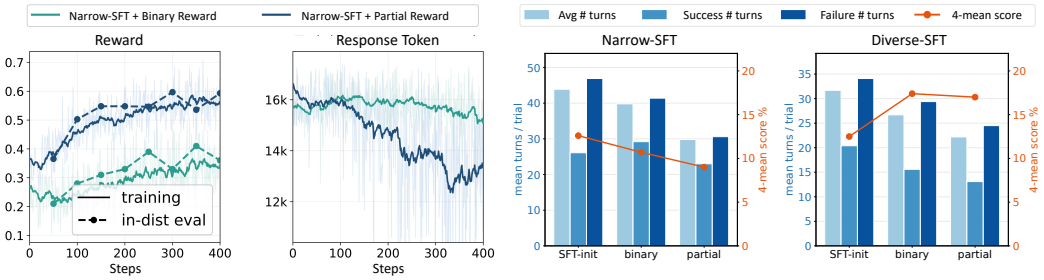


Figure 11: Training curves and evaluation results for different SFT initializations (Narrow-SFT vs. Diverse-SFT) and trajectory-level reward designs (binary vs. partial). All RL experiments are conducted using TermiGen (Zhu et al., 2026).

Using **Diverse-SFT** mitigates these reward-hacking risks. As shown in Fig. 11 (b, right), when starting from Diverse-SFT while keeping all other configurations unchanged, the evaluation performance after RL improves under both the binary-reward and partial-reward settings. Our interpretation is that exposure to diverse atomic skills equips the model with the capabilities needed to solve the training tasks, making it less likely to exploit shortcuts during policy optimization and thereby mitigating reward hacking to some extent.

[3]TermiGen environments were used for SFT data collection in their experiments.