REVISITING DAGGER IN THE ERA OF LLM-AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Long-horizon LM agents learn from multi-turn interaction, where a single early mistake can alter the subsequent state distribution and derail the whole trajectory. Existing recipes fall short in complementary ways: supervised fine-tuning provides dense teacher supervision but suffers from *covariate shift* because it is trained on off-policy teacher trajectories; while reinforcement learning with verifiable rewards avoids this off-policy mismatch by learning from on-policy rollouts but with only *sparse outcome feedback*. We address this dilemma by revisiting Dataset Aggregation (DAgger) for multi-turn LM agents: the algorithm collects trajectories through a turn-level interpolation of student and teacher policies, and the student is then trained on these trajectories using supervised labels provided by the teacher. By directly interacting with environments, we expose the model to realistic states likely to be encountered during deployment, thereby effectively mitigating covariate shift. Besides, since the student is learned by mimicking the teacher's behavior, it receives rich feedback during learning. To demonstrate DAgger enjoys the benefits of both worlds, we tested the algorithm to train a software-engineering agent with 4B- and 8B-scale student models. On SWE-bench Verified, our DAgger-style training improves over the strongest post-training baseline by $+3.9$ points at 4B and $+3.6$ points at 8B. The resulting 4B agent reaches 27.3%, outperforming representative published 8B SWE-agent systems, while the 8B agent achieves 29.8%, surpassing SWE-Gym-32B and coming within 5 points of stronger 32B-scale agents. Together with consistent gains on the held-out SWE-Gym split, these results suggest the effectiveness of DAgger for modern long-horizon LM agents.

1 INTRODUCTION

Large language models (LLMs) are increasingly deployed as interactive agents that operate over long horizons: they call tools, observe environment feedback, and make decisions across many turns. This agentic setting is central to emerging applications such as software-engineering agents for resolving real GitHub issues (Jimenez et al., 2023; Pan et al., 2024; Hou et al., 2024), web-browsing agents (Wei et al., 2025a; Chen et al., 2025; Avraham et al., 2026), AI research scientist (Qiang et al., 2025a;b; Chen et al., 2026) and general-purpose tool-using assistants (Chen et al., 2024; Wang et al., 2024b), which urges the development of efficient post-training algorithms for agentic setting.

Despite the apparent success of the post-training techniques, it remains unclear how to efficiently post-train these agents for such multi-turn, long-context tasks. In fact, each of the existing recipes has structural limitations that become pronounced in long-horizon agentic tasks. While *supervised fine-tuning (SFT)*, which directly imitates teacher trajectories (Ouyang et al., 2022; Yang et al., 2025b; Pan et al., 2024; Qiang et al., 2025a), provides dense, token-level supervision, it trains the policy exclusively on expert-induced states. This leads to *covariate shift*: during deployment, prefix states are sampled by the student, where early errors can cause significant divergence in the state distribution and degrade performance. *Reinforcement learning with verifiable rewards (RLVR)* (Shao et al., 2024; Wei et al., 2025b; Yang et al., 2025c; Schulman et al., 2017) addresses distribution mismatch by training over the student's own rollouts with the outcome-level rewards through policy-gradient. It suffers from sparse credit assignment, typically providing only a single outcome-level reward for an entire trajectory (Shao et al., 2024). Furthermore, RL is computationally expensive due to group sampling, and advantage estimates collapse when samples lack diversity in correctness Yu et al. (2025); Yang et al. (2025d). *On-policy distillation (OPD)* (Agarwal et al., 2024; Zhao et al., 2026; Hübotter et al., 2026; Yang et al., 2025a) trains the student to match the teacher's token-level distributions on student-generated trajectories, combining on-policy state coverage with dense

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.

Table 1: Comparison of post-training recipes for multi-turn LM agents.

	SFT	RLVR	On-Policy Distillation	Ours
On-Policy Data	✗	✓	✓	✓
Dense Learning Signal	✓	✗	✓	✓
Low Sampling Cost	✓	✗	✓	✓
Cold-Start Robustness	✓	✗	✗	✓
Compatibility with Black-Box Teacher	✓	✗	✗	✓

teacher supervision. However, OPD yet faces a cold-start bottleneck: early rollouts from weak students often fail prematurely, especially with long horizon tasks, forcing the teacher to supervise unsuccessful prefixes rather than productive trajectories (Agarwal et al., 2024). Meanwhile, OPD requires teacher token probabilities, which black-box LLMs such as Gemini (Google DeepMind, 2026) and GPT (OpenAI, 2026) do not expose. Extending OPD to this setting requires additional machinery (Ye et al., 2025). This motivates a central question:

Is there any method that can simultaneously exploit dense feedback, while with on-policy coverage and early access to successful trajectories?

In response to this challenge, we revisit Dataset Aggregation (DAgger) (Ross et al., 2011) for LLM-based agents, a classical imitation-learning algorithm designed to reduce covariate shift in sequential decision making. The key idea of DAgger is to gradually supervise the student using states visited by the student itself, and we adapt this principle to multi-turn LM agents through *teacher-interleaved trajectory collection*. Specifically, each trajectory is generated via a stochastic mixture of student and teacher turns; while student actions expose the model to deployment-accurate states, periodic teacher takeovers ensure trajectories reach productive outcomes. The probability of teacher intervention gradually decays throughout training. We then train the student on these trajectories to mimick teacher behaviors, therefore, learning from dense feedback while mitigating the covariate shift inherent in SFT. Furthermore, because teacher actions dominates in early trajectories and are involved in the whole training procedure, this approach avoids the unnecessary exploration in the cold-start failure mode typical of OPD, and therefore, is more sample efficient.

We demonstrate that this design is especially well-suited to software-engineering (SWE) tasks, where an agent must operate inside a codebase over many turns, searching files, localizing bugs, editing code, and submitting a patch. Early mistakes can derail the interaction, leading to states where expert demonstrations provide no coverage and student-only rollouts fail to recover. In this setting, teacher-interleaved DAgger provides a practical post-training recipe that combines on-policy state coverage with teacher-guided recovery and dense supervision, directly targeting the failure modes of SFT, RLVR, and OPD. Empirically, our method delivers strong gains at both 4B and 8B scales: the 4B agent surpasses representative 8B SWE agents, while the 8B agent approaches stronger 32B-scale systems. Beyond final task resolution, our analyses show that DAgger stabilizes training, mitigates covariate shift, and improves long-horizon agent behaviors such as search, editing, and recovery.

2 Preliminaries

Behavior cloning and covariate shift. Consider a finite-horizon sequential decision problem with horizon T, state space $\mathcal{S}$, action space $\mathcal{A}$, a student policy π_θ, and a teacher (expert) policy π_e. Let d_π^t denote the state distribution induced at step t by rolling out policy π, and let $d_\pi = \frac{1}{T}\sum_{t=1}^{T} d_\pi^t$ be the corresponding average state distribution. Behavioral cloning trains the learner by minimizing a supervised loss on states drawn from the teacher distribution:

$$\min_\theta \ \mathbb{E}_{s\sim d_{\pi_e},\, a_e\sim\pi_e(\cdot|s)} \left[\ell\big(\pi_\theta(\cdot \mid s), a_e\big)\right], \tag{1}$$

where ℓ is typically cross-entropy loss for discrete actions. Despite its simplicity, this objective trains exclusively on teacher-induced states. During deployment, however, the student follows its own distribution d_{π_θ}, where compounding prediction errors can shift trajectories outside the training support. In the worst case, this covariate shift causes imitation error to scale quadratically with the horizon T (Ross & Bagnell, 2010; Ross et al., 2011).

Dataset Aggregation (DAgger) and AggreVaTe. DAgger (Ross et al., 2011) addresses this mismatch by training on student-visited states. At iteration i, trajectories follow a mixture policy

$$\mu_i = \beta_i \pi_e + (1 - \beta_i)\pi_{\theta_i}, \tag{2}$$

where $\beta_i \in [0, 1]$ is typically annealed toward zero across iterations. For each state s visited under μ_i, DAgger queries the teacher for an action $a_e \sim \pi_e(\cdot \mid s)$ and aggregates the pairs into a dataset

$$\mathcal{D}_{i+1} = \mathcal{D}_i \cup \{(s, a_e) : s \sim d_{\mu_i},\ a_e \sim \pi_e(\cdot \mid s)\}. \tag{3}$$

The next learner is then obtained by supervised learning on the aggregated dataset:

$$\theta_{i+1} = \arg\min_{\theta} \mathbb{E}_{(s,a_e)\sim\mathcal{D}_{i+1}} \left[\ell\big(\pi_\theta(\cdot \mid s), a_e\big)\right]. \tag{4}$$

The key distinction from behavioral cloning lies in the state distribution: DAgger trains the learner on the states it is actually likely to visit. By doing so, DAgger provides no-regret guarantees and improves the imitation error's dependence on horizon from quadratic to linear (Ross et al., 2011).

AggreVaTe (Ross & Bagnell, 2014) builds on this by employing a distinct sampling protocol: the student policy generates an initial trajectory prefix, after which a teacher takes over to complete the sequence from a specific intervention point. We also adopt this *student-prefix, teacher-completion* protocol as one option of our sampling strategy.

3 METHODS

3.1 MULTI-TURN LM-AGENT SETTING

We first set up the notation of the multi-turn LM-agent setting. A task instance is specified by an initial prompt $x \sim q$ drawn from a task distribution q, such as a software issue or a web-browsing query. Given x, the agent interacts with an environment over a sequence of turns. At turn t, the policy observes the interaction history and samples an action a_t, which may include intermediate reasoning and a tool invocation. The environment then executes the action and returns an observation o_t:

$$a_t \sim \pi(\cdot \mid x, a_{1:t-1}, o_{1:t-1}), \qquad o_t \sim \mathrm{Env}(\cdot \mid x, a_{1:t}, o_{1:t-1}). \tag{5}$$

The interaction terminates when the agent emits a designated `finish` action or reaches a maximum turn budget $T_{\max}$, producing a trajectory:

$$\tau = (x, a_1, o_1, \ldots, a_T, o_T), \qquad T \le T_{\max}. \tag{6}$$

A verifier then assigns a terminal success signal $R(\tau, x) \in \{0, 1\}$, indicating whether the trajectory solves the task. For example, in software-engineering tasks, R may stands for whether the final patch passes the relevant tests; we defer the concrete instantiation to Section 4.

For compactness, we define the *states* as the observable interaction history:

$$s_t \triangleq (x, a_{1:t-1}, o_{1:t-1}). \tag{7}$$

Thus, the policy can be written as $\pi(\cdot \mid s_t)$. Throughout, π_e denotes a stronger teacher policy and π_θ the student policy to be trained. Our goal is to improve π_θ with supervision from π_e.

3.2 DAGGER FOR MULTI-TURN LM AGENTS

In this section, we detail our rollout protocols and training objectives, which adapt the DAgger principle for the post-training of multi-turn LM agents. We propose two distinct rollout strategies that both integrate student and teacher sampling to expose the model to states likely encountered during deployment. Throughout these rollouts, teacher labels and executed actions are collected and saved at each turn, which are then used to optimize the student policy via a per-turn combination of cross-entropy and on-policy distillation objectives.

Rollout with Stochastic Policy Mixture. For each turn given history s_t, we define a binary indicator $b_t \in \{0, 1\}$ to determine whether to execute an action a_e from the teacher policy $\pi_e(\cdot \mid s_t)$ (if $b_t = 1$) or an action a_θ from the student policy $\pi_\theta(\cdot \mid s_t)$ (if $b_t = 0$). We propose two distinct protocols for determining these indicators across a trajectory.

i) *DAgger-style Rollout:* At iteration i, we define a mixing parameter $\beta_i \in [0, 1]$, which is decayed towards 0 across iterations. The sequence of indicators for a trajectory is sampled according to:

$$p_i^{\text{turn}}(b_{1:T_{\max}}) = \prod_{t=1}^{T_{\max}} \beta_i^{b_t}(1-\beta_i)^{1-b_t}. \tag{8}$$

This is a *turn-level* mixture: each turn's executor is drawn independently with probability β_i.

ii) *AggreVaTe-style Rollout:* At iteration i, we define a distribution ρ_i over $\{0, \ldots, T_{\max}\}$. For each trajectory, we sample a student-prefix length $\kappa \sim \rho_i$ and set the indicators as follows:

$$p_i^{\text{traj}}(b_{1:T_{\max}} \mid \kappa) = \prod_{t=1}^{\kappa} \mathbb{I}\{b_t = 0\} \prod_{t=\kappa+1}^{T_{\max}} \mathbb{I}\{b_t = 1\}. \tag{9}$$

This represents a *trajectory-level* mixture, where the student maintains control until timestep κ, after which the teacher completes the rollout. We schedule ρ_i so that student prefixes grow over training, gradually shifting AggreVaTe-style rollouts toward the on-policy distribution.

At every visited state s_t, we query the student or the teacher according to b_t and execute

$$a_t \sim \begin{cases} \pi_e(\cdot \mid s_t), & b_t = 1, \\ \pi_\theta(\cdot \mid s_t), & b_t = 0. \end{cases} \tag{10}$$

The environment returns $o_t \sim \text{Env}(\cdot \mid x, a_{1:t}, o_{1:t-1})$, and the rollout terminates when the agent emits `finish` or reaches the turn limit $T_{\max}$. Crucially, regardless of which action we execute, we will query the expert action $\tilde{a}_t \sim \pi_e(\cdot \mid s_t)$ in every visited state. Together with the execution trace, they contribute the final data batch used for training:

$$\mathcal{B}(\tau) = \{(s_t, \tilde{a}_t) : t = 1, \ldots, T\}, \tag{11}$$

i.e., the prefix comes from the execution trace, while the label is provided by the expert. Algorithm 1 summarizes the procedure. Overall, both rollout schedules implement the same principle: early training benefits from teacher-guided trajectories and recovery, while later training increasingly exposes the student to its own deployment-time state distribution.

Training Objective. Following the rollout in the i-th iteration, we use the logged data $\mathcal{D}_i = \{(s_t, \tilde{a}_t)\}_{t=1}^{N}$ to train the student with a cross-entropy loss against the expert labels:

$$\mathcal{L}_i(\theta) = \mathbb{E}_{(s_t, \tilde{a}_t) \sim \mathcal{D}} \left[\sum_{t=1}^{T} \ell_{\text{CE}}(\theta; s_t, \tilde{a}_t) \right]. \tag{12}$$

For an expert action $\tilde{a}_t$ consisting of m_t tokens $(\tilde{a}_{t,1}, \ldots, \tilde{a}_{t,m_t})$, the cross-entropy loss is defined as:

$$\ell_{\text{CE}}(\theta; s_t, \tilde{a}_t) = -\sum_{j=1}^{m_t} \log \pi_\theta \left(\tilde{a}_{t,j} \mid s_t, \tilde{a}_{t,<j} \right). \tag{13}$$

For computational efficiency, we pack transitions with shared prefix into one single trajectory and applying a loss mask to ensure the gradient remains equivalent to the individual update.

Executor-Matched Supervision. When teacher token log-probabilities are available, we additionally consider executor-matched supervision. The objectives above label every visited state with the teacher action under a uniform cross-entropy weight, regardless of which policy executed that turn. A natural generalization decouples the supervision target from the executor, allowing the loss at each turn to follow the policy that actually produced it. Let a_t denote the action executed at the realized prefix s_t, with turn log-probability $\log \pi_\theta(a_t \mid s_t) = \sum_j \log \pi_\theta(a_{t,j} \mid s_t, a_{t,<j})$ factorized over tokens as in Eq. (12). This gives

$$\mathcal{L}_i^{\text{EM}}(\theta) = \mathbb{E}_{\tau \sim \eta_i} \left[\sum_{t=1}^{T} -\,\text{sg}(w_t)\, \log \pi_\theta(a_t \mid s_t) \right], \qquad w_t = \begin{cases} 1, & b_t = 1, \\ -\log \dfrac{\pi_\theta(a_t \mid s_t)}{\pi_e(a_t \mid s_t)}, & b_t = 0, \end{cases} \tag{14}$$

where η_i is the mixture-rollout distribution of Algorithm 1 and $\text{sg}(\cdot)$ stops the gradient through the weight. The construction interpolates between two supervision regimes according to the executor indicator b_t. A teacher-executed turn ($b_t = 1$) reduces to cross-entropy on the teacher action, recovering Eq. (12); a student-executed turn ($b_t = 0$) supervises the student's own action under the

Algorithm 1 Stochastic Mixing Rollout Collection with Teacher Labels.

Require: Teacher π_e, student π_θ, prompt x, regime $r \in \{\textsc{Turn}, \textsc{Traj}\}$, β_i, ρ_i, horizon $T_{\max}$
1: Sample $b_{1:T_{\max}} \sim p_i^{\text{turn}}$ as in Eq. (8) if $r = \textsc{Turn}$
2: Otherwise, sample $\kappa \sim \rho_i$ and $b_{1:T_{\max}} \sim p_i^{\text{traj}}(\cdot \mid \kappa)$ as in Eq. (9)
3: $s_1 \leftarrow x$
4: **for** $t = 1, \ldots, T_{\max}$ **do**
5: Query teacher label $\tilde{a}_t \sim \pi_e(\cdot \mid s_t)$
6: Set $a_t \leftarrow \tilde{a}_t$ if $b_t = 1$; otherwise sample $a_t \sim \pi_\theta(\cdot \mid s_t)$
7: Sample $o_t \sim \text{Env}(\cdot \mid x, a_{1:t}, o_{1:t-1})$ and set $s_{t+1} \leftarrow (x, a_{1:t}, o_{1:t})$
8: **if** $a_t = \texttt{finish}$ **then break**
9: **end for**
10: **return** $\tau = (x, a_{1:T}, o_{1:T})$, $b_{1:T}$, and $\mathcal{B}(\tau) = \{(s_t, \tilde{a}_t)\}_{t=1}^{T}$

reverse-KL distillation weight. Crucially, this weighting is orthogonal to the rollout protocol: both the DAgger-style and AggreVaTe-style schedules admit either supervision rule. The state distribution is thus controlled by the rollout schedule and the supervision by the per-turn weight, yielding a two-axis design space that we formalize in Section 3.3.

3.3 A Unified Perspective on Post-Training Algorithms

In this section, we situate our DAgger algorithm within a unified framework alongside other post-training methods, such as SFT, On-policy Distillation, and RL. Notably, the training objectives for all these algorithms can be described through a unified language:

$$\theta_{i+1} = \arg\max_{\theta} \mathbb{E}_{s\sim \text{sg}(p_s), a\sim \text{sg}(p_a)} \left[\text{sg}(w(s,a)) \log \pi_\theta(a \mid s)\right] - \lambda \Omega_i(\theta) \tag{15}$$

where $\text{sg}(\cdot)$ denotes gradient stopping. In this formulation, s represents the context sampled from the context distribution p_s, a denotes the turn-level action label drawn from the label distribution $p_a(\cdot \mid s)$, $w(s,a)$ serves as a scoring function that weights the importance of each sample, and Ω_i is an optional regularizer (e.g., KL divergence) used to constrain the update.

Table 2: Unified view of post-training methods by their choice of p_s, p_a, and $w(s,a)$. In the last row, paired labels and weights correspond to teacher- and student-executed turns, respectively.

Method	**Context Dist.** p_s	**Label Dist.** p_a	**Scoring Function** $w(s,a)$
SFT / BC	d_{π_e}	π_e	$w(s,a) \equiv 1$
RL (Policy Gradient)	d_{π_θ}	π_θ	$w(s,a) = A(s,a)$
OPD	d_{π_θ}	π_θ	$w(s,a) = -\log \frac{\pi_\theta(a\|s)}{\pi_e(a\|s)}$
Ours	d_i^{turn} / d_i^{traj}	π_e	1
Ours *w/ Executor-Matched Supervision*	d_i^{turn} / d_i^{traj}	π_e / π_θ	1 / $-\log \frac{\pi_\theta(a\|s)}{\pi_e(a\|s)}$

We provide a derivation of this unified objective and a detailed mapping of each algorithm to the choices of p_s, p_a, and $w(s,a)$ in Appendix A. This unified perspective allows for a rigorous comparison of post-training methodologies. As shown in Table 2, SFT represents the most straightforward instantiation, where both the context and label distributions are derived solely from the expert policy π_e with a uniform scoring function $w(s,a) \equiv 1$. In contrast, RL and OPD sample both trajectories via the current policy π_θ and utilize scoring functions based on advantages or log-likelihood ratios to prioritize high-value actions. Our DAgger-style and AggreVaTe-style approaches bridge these paradigms. They employ decaying context distributions (d_i^{turn} and d_i^{traj}) induced by their rollout policies (p_i^{turn} and p_i^{traj}), which interpolates between student and teacher distributions to effectively mitigate covariate shift. By default, both rollout variants use teacher labels with unit weights. With optional executor-matched supervision (Eq. (14)), teacher-executed turns retain cross-entropy, while student-executed turns use student actions with reverse-KL distillation weights.

4 Experiments

We conduct comprehensive experiments to answer the following research questions:

1. **Effectiveness.** How does our method compare with SFT, GRPO, and on-policy distillation in task-resolution rate? (§4.2; results across multiple evaluation seeds in Appendix F.4)
2. **Training stability.** Under a matched compute budget, does our method produce a more stable and consistently improving training trajectory than competing post-training methods? (§4.3; results under various rollout schedules in Appendix F.1)
3. **Covariate shift.** Does our method reduce trajectory-level distribution shift at deployment? (§4.4; additional divergence measures: Appendix F.5)
4. **Agent behavior.** How does our method change agent behavior beyond resolution rates? (§4.5)
5. **Generalization.** Do our findings hold across teachers and task domains? (Appendix F.2)
6. **Loss design.** Under identical mixture rollouts, how does executor-matched supervision compare with pure cross-entropy? (Appendix F.3)

4.1 Experimental Setup

Models and datasets. We instantiate the student policy π_θ with two model scales from the Qwen3 family (Yang et al., 2025a): Qwen3-4B-Instruct-2507 and Qwen3-8B. Across all configurations, we use Qwen3-Coder-30B-A3B-Instruct (Cao et al., 2026) as the fixed teacher policy π_e. All training is performed on SWE-Gym (Pan et al., 2024), a collection of real-world software-engineering tasks paired with executable unit-test suites. For in-domain evaluation, we reserve a fixed set of 100 SWE-Gym instances as a held-out split, which we refer to as SWE-Gym Holdout, and train on the remaining $2{,}338$ tasks. For out-of-domain evaluation, we report results on SWE-Bench Verified (Chowdhury et al., 2024)[1], following the standard task-resolution metric. Dataset details are given in Appendix D.

Baselines. We compare against two sets of baselines. First, we consider three post-training methods trained on SWE-Gym from the same student initialization: (1) **SFT** uses teacher-generated expert trajectories from the initial prompt, following the SWE-Gym training recipe (Pan et al., 2024), with rejection sampling based on executable test feedback; (2) **GRPO** follows the SkyRL-v0 recipe (Cao et al., 2025; Zhang et al., 2026): the student samples a group of eight trajectories per task under its own policy and is updated by policy gradient with executable test outcomes as the terminal reward; (3) **On-policy distillation** follows (Lu & Thinking Machines Lab, 2025): the student collects trajectories under its own policy, while the teacher supplies token-level supervision at student-visited states through a reverse-KL distillation objective. Second, to place our results in the broader SWE-agent literature, we also report published SWE-Bench Verified resolution rates from representative SWE-agent systems (Yang et al., 2025b; Cao et al., 2025; Pan et al., 2024; Zhu et al., 2025; Wang et al., 2025).

Agent Scaffolding. All trajectories are generated and evaluated with OpenHands (Wang et al., 2024b), including its tool interface and execution environment. To ensure fair comparison across model families, we canonicalize trajectories during data construction and re-render them into each model's native chat and tool-use template during training. Details are provided in Appendix G.

Implementation Details. Unless otherwise specified, our two variants share the optimization and rollout-update budget of the baselines. At each iteration, we collect a fresh mixture-rollout batch, update the student on teacher-labeled data, and evaluate with greedy decoding. *In the main experiments, both rollout variants use cross-entropy on sampled teacher actions*, while executor-matched supervision can yield further gains when teacher log-probabilities are available (Appendix F.3). Rollout schedules, sampling parameters, context limits, and hyperparameters are listed in Appendix E.

4.2 Main Results

Table 3 reports the main results on SWE-Gym Holdout and SWE-Bench Verified. Under the matched OpenHands scaffold and SWE-Gym training data, our DAgger-style training consistently outperforms prior post-training methods across both student scales. For Qwen3-4B-Instruct-2507, DAgger-style training achieves 17.0% on SWE-Gym Holdout and 27.3% on SWE-Bench Verified, improving over the strongest non-DAgger baseline, OPD, by $+1.0$ and $+3.9$ points, respectively. For Qwen3-8B, the gains are larger: DAgger-style training reaches 19.0% and 29.8%, exceeding OPD by $+3.0$ points

[1] 34 Matplotlib tasks in SWE-Bench Verified (500 tasks) fail to build in our Docker environment, so we report rates on the remaining 466; spot checks put the effect on the aggregate rate below 1%.

Table 3: Main results on SWE-Gym Holdout and SWE-Bench Verified (task-resolution rate; higher is better). Top: post-training methods under matched student initialization and scaffold; bottom: published SWE-agent systems. Best and second-best per backbone block in **bold** and underline.

Method	Scaffold	Data	SWE-Gym Holdout	SWE-Bench Verified
Post-training Methods				
Qwen3-4B	OpenHands	–	5.0%	11.2%
+ GRPO	OpenHands	SWE-Gym	9.0%	15.6%
+ SFT	OpenHands	SWE-Gym	15.0%	22.9%
+ OPD	OpenHands	SWE-Gym	16.0%	23.4%
+ Ours (*DAgger-style*)	OpenHands	SWE-Gym	**17.0%**	**27.3%**
+ Ours (*AggreVate-style*)	OpenHands	SWE-Gym	16.0%	24.5%
Qwen3-8B	OpenHands	–	2.0%	7.7%
+ GRPO	OpenHands	SWE-Gym	8.0%	13.2%
+ SFT	OpenHands	SWE-Gym	12.0%	23.4%
+ OPD	OpenHands	SWE-Gym	16.0%	26.2%
+ Ours (*DAgger-style*)	OpenHands	SWE-Gym	**19.0%**	**29.8%**
+ Ours (*AggreVate-style*)	OpenHands	SWE-Gym	17.0%	27.3%
Published SWE-agent systems				
∼7–8B backbones				
SWE-Gym-7B (Pan et al., 2024)	OpenHands	SWE-Gym	–	10.6%
SkyRL-Agent-7B-v0 (Cao et al., 2025)	OpenHands	SkyRL-v0	–	14.6%
SkyRL-Agent-8B-v0 (Cao et al., 2025)	OpenHands	SkyRL-v0	–	9.4%
SWE-smith-LM-7B (Yang et al., 2025b)	SWE-Agent	SWE-smith	–	15.2%
R2E-Gym-7B-Agent (Zhu et al., 2025)	OpenHands	R2E-Gym	–	19.0%
∼32B backbones				
SWE-Gym-32B (Pan et al., 2024)	OpenHands	SWE-Gym	–	20.6%
R2E-Gym-32B-Agent (Zhu et al., 2025)	OpenHands	R2E-Gym	–	34.4%
SWE-Dev-32B (Wang et al., 2025)	SWE-Dev	SWE-Dev	–	36.6%

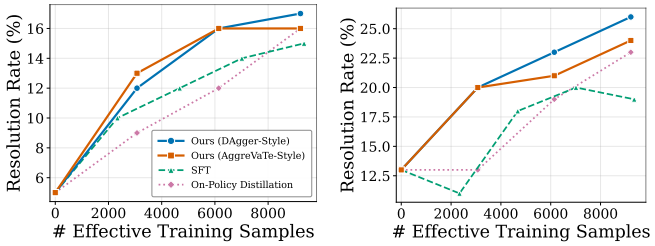
on SWE-Gym Holdout and $+3.6$ points on SWE-Bench Verified. The AggreVaTe-style variant also improves over SFT and GRPO, and remains competitive with OPD, indicating that teacher-completion rollouts provide useful supervision even with a simpler trajectory-level intervention scheme. GRPO underperforms teacher-based methods despite higher compute usage, with sparse rewards limiting learning: 80–90% of task groups yield only zero rewards in both the 4B and 8B runs.

We also compare against published SWE-agent systems in the lower block of Table 3. Although these systems differ in training data and, in some cases, scaffolding, the comparison contextualizes the strength of our post-training recipe. Notably, our 4B DAgger-style model achieves 27.3% on SWE-Bench Verified, outperforming the published SkyRL-Agent-8B-v0 result by $+17.9$ points and the strongest published 7B-scale result in the table, R2E-Gym-7B-Agent, by $+8.3$ points. Moreover, our 8B DAgger-style model reaches 29.8% on SWE-Bench Verified, surpassing SWE-Gym-32B by $+9.2$ points and narrowing the gap to stronger 32B agents, trailing R2E-Gym-32B-Agent and SWE-Dev-32B by only 4.6 and 6.8 points, respectively. These results suggest that adapting DAgger-style state-distribution correction to multi-turn LM agents can yield substantial gains beyond standard SFT, RL, and on-policy distillation baselines, enabling smaller backbones to approach the performance of substantially larger SWE-agent systems.

4.3 Sample Scaling and Training Stability

We next study training-data scaling under matched effective-sample budgets in the 4B setting, using *Qwen3-4B-Instruct-2507* as the student. Figure 1 compares our *DAgger-style* and *AggreVaTe-style* variants against SFT with rejection sampling and on-policy distillation on SWE-Gym Holdout and a fixed 100-task SWE-Bench Verified subset, which we verified closely tracks the full benchmark. We omit GRPO from the scaling curves because it uses a different rollout budget, sampling eight trajectories per task, while also yielding smaller gains than teacher-based methods (Table 3).

We find that teacher-interleaved training yields a more stable scaling trajectory. At 3K effective samples, DAgger-style training reaches 12% on SWE-Gym Holdout and 20% on SWE-Bench Verified-100, outperforming on-policy distillation at 9% and 13%; AggreVaTe-style shows a similar



(a) SWE-Gym Holdout (b) SWE-Bench Verified-100

Figure 1: Scaling with effective training samples for the 4B student on SWE-Gym Holdout and SWE-Bench Verified-100; higher is better.

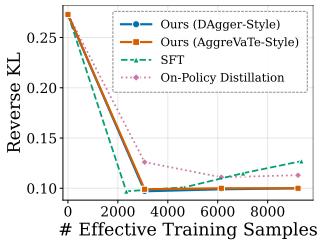


Figure 2: Policy divergence under student-induced rollouts (4B student): average token-level reverse KL $D_{\mathrm{KL}}(\pi_\theta \| \pi_e)$ on student-visited contexts; lower is better.

Table 4: Qualitative failure analysis on SWE-bench Verified. An LLM judge assigns one primary failure mode to each unresolved trajectory. Resolve and submission rates are over all 466 instances; failure rates are within their unresolved subsets.

Model / Method	Resolve	Submission		Submitted but Unresolved			No Submission				
				Wrong	Wrong	Syntax or	Tool-Use	Repetitive	Context	Budget	No
	(%)	Yes	No	Solution	File	Runtime Error	Error	Loop	Overflow	Exhaustion	Edit
Qwen3-4B-Instruct-2507	11.2%	52.1%	47.9%	74.3%	21.1%	6.6%	3.8%	16.0%	29.4%	49.2%	5.3%
+ GRPO	15.6%	53.4%	46.6%	72.3%	24.9%	2.8%	3.4%	14.5%	28.1%	48.9%	8.5%
+ SFT	22.9%	97.6%	2.4%	55.8%	23.8%	20.3%	1.1%	30.5%	55.1%	13.9%	0.5%
+ On-Policy Distillation	23.4%	98.5%	1.5%	63.2%	19.1%	17.6%	2.3%	44.3%	15.4%	40.3%	0.0%
+ Ours (*DAgger-Style*)	27.3%	97.9%	2.1%	64.4%	19.7%	15.9%	0.0%	22.7%	57.5%	19.3%	0.0%
+ Ours (*AggreVaTe-Style*)	24.5%	97.2%	2.8%	66.4%	15.7%	17.9%	1.3%	23.7%	60.1%	16.2%	0.0%

early advantage at 13% and 20%. This supports our cold-start motivation: student-only rollouts often enter unproductive states early, whereas teacher interleaving provides successful recoveries and dense supervision from the beginning. At larger budgets, DAgger-style continues improving to 17% and 26%, while SFT reaches only 15% on SWE-Gym Holdout and peaks at 20% before dropping to 19% on SWE-Bench Verified-100. This supports our covariate-shift motivation: SFT trains only on expert-induced states, whereas DAgger gradually shifts supervision toward student-induced states while retaining expert corrections. Together, these trends show that mixture rollouts provide both a stronger cold start than OPD and better asymptotic scaling than SFT.

4.4 Policy Divergence under Student-Induced Rollouts

We next examine whether teacher-interleaved training reduces deployment-time covariate shift. In the 4B setting, we sample 100 SWE-Gym instances, generate one student rollout per instance with temperature 0.7, and compute the average token-level reverse KL, $\mathbb{E}_{c\sim\hat{d}^{\pi_\theta}}[D_{\mathrm{KL}}(\pi_\theta(\cdot|c)\|\pi_e(\cdot|c))]$, on contexts visited by the student. Lower reverse KL indicates that the student remains closer to the teacher on its own deployment-time state distribution.

Figure 2 shows that SFT exhibits a clear covariate-shift pattern: its reverse KL initially drops from roughly 0.27 to 0.10, but later rebounds to about 0.126 as training continues, suggesting that expert-only trajectories fail to cover the states induced by the learned policy. In contrast, both DAgger-style and AggreVaTe-style training remain nearly flat around 0.10 after the initial drop, yielding roughly a 20% reduction relative to SFT at the largest sample budget. Compared with on-policy distillation, our methods also reduce divergence earlier and more stably: around 3K effective samples, OPD remains near 0.126, while both mixture-rollout variants are already close to 0.10. These trends support the two intended effects of teacher interleaving: early teacher actions mitigate OPD's cold-start rollouts, while later student-induced states reduce the train-test mismatch of SFT.

4.5 Qualitative Failure Analysis

We further analyze how post-training changes agent behavior beyond aggregate resolution rates. Following SWE-Agent (Yang et al., 2024), we use *Claude Opus 4.7* as an LLM judge to assign one primary failure mode to each unresolved trajectory on SWE-Bench Verified, separating failures into submitted-but-unresolved and no-submission cases.

Table 4 shows that GRPO improves task-resolution rate over the base model (15.6% vs. 11.2%), with similar submission rates (53.4% vs. 52.1%). SFT and on-policy distillation largely learn the submission format, raising submission rates to 97.6% and 98.5%, but still suffer from unstable failure modes: SFT has the highest syntax/runtime error rate among trained methods (20.3%), while on-policy distillation exhibits a high repetitive-loop rate among no-submission failures (44.3%). In contrast, DAgger-style training achieves the best resolution rate (27.3%), maintains a high submission rate (97.9%), and reduces syntax/runtime errors to 15.9%, indicating that mixture rollouts improve not only whether the agent submits but also the quality of submitted patches.

The remaining failures further clarify the effect of teacher interleaving. DAgger-style no-submission failures are dominated by context overflow (57.5%), rather than no-tool-use or budget exhaustion, suggesting that the main residual bottleneck is long-context capacity rather than failure to make progress. AggreVaTe-style training achieves the lowest wrong-file rate (15.7%), suggesting that longer teacher continuations can improve repository localization, although its overall resolution rate remains below DAgger-style training. Overall, mixture-policy training shifts failures from passive or malformed behavior toward long-horizon limitations such as localization and context management.

5 Related Works

Coding Agent and SWE Tasks Large language models (LLMs) have shown strong potential in Software Engineering (SE) tasks (Liu et al., 2024), where they have demonstrated encouraging capabilities across a range of tasks, including code generation (Zhang et al., 2024; Yetiştiren et al., 2023), software testing (Wang et al., 2024a), and automated debugging (Xia et al., 2023). More recently, LLMs have been embedded into agentic frameworks that reason over goals, interact with external tools, execute commands, and modify code in realistic repositories.

Within SWE, SWE-Bench (Jimenez et al., 2023) introduced repository-level issue resolution based on real GitHub issues and unit tests, revealing the difficulty of long-horizon software maintenance tasks. Subsequent benchmarks and training environments (Pan et al., 2024; Yang et al., 2025b; Badertdinov et al., 2025; Zan et al., 2025; Deng et al., 2025), together with agent scaffolds such as SWE-agent (Yang et al., 2024) and OpenHands (Wang et al., 2024b) demonstrate that agents equipped with developer-like interactions—such as file navigation, code editing, command-line execution, and testing—can more effectively operate in realistic software projects.

LLM Agent Post Training Post-training LLM agents with verifiable rewards has become a prominent alternative to preference-based supervision, especially in SWE tasks where unit tests provide automatic outcome signals (Pan et al., 2024; Guo et al., 2024). However, such rewards are sparse and provide limited credit assignment over long trajectories, motivating denser forms of supervision such as rubric-based feedback (Gunjal et al., 2025; Huang et al., 2026) and on-policy distillation (OPD) (Lu & Thinking Machines Lab, 2025; Agarwal et al., 2024; Gu et al., 2024). OPD improves over offline SFT by training on student-induced states, but still relies on student-generated trajectories, which can be brittle under cold-start failures. Our method revisits DAgger (Ross et al., 2011) for LM agents, interpolating between expert-guided imitation and on-policy state coverage through teacher-interleaved rollouts. Lauffer et al. (2025) use a random student-to-expert switch, a special case of our unified view without annealing or turn-level mixing; black-box OPD (Ye et al., 2025) requires adversarial training, whereas our cross-entropy objective needs only sampled teacher actions.

6 Conclusion

We revisit DAgger for multi-turn LM agents, motivated by the train-test state-distribution mismatch that arises when local errors compound through long-horizon tool interactions. Through teacher-interleaved mixture rollouts, our method combines on-policy state coverage with teacher-guided recovery and dense supervised feedback. Experiments on software-engineering agents show consistent gains over SFT, GRPO, and on-policy distillation at both 4B and 8B scales, with the 4B agent surpassing representative 8B SWE agents and the 8B agent approaching stronger 32B-scale systems. Further analyses show that these gains arise from more stable training, reduced deployment-time policy mismatch, and improved long-horizon behaviors such as search, editing, and recovery. Our results highlight explicit state-distribution correction as a powerful ingredient for post-training LM agents that interact with environments over increasingly long horizons.

AI USE STATEMENT

Language models were used to generate training trajectories and teacher supervision for the post-training experiments (Section 4.1), and Claude Opus 4.7 was used to classify failures in unresolved SWE-Bench Verified trajectories (Section 4.5). Software-engineering task resolution was determined by the benchmark's executable tests. We also used AI tools to assist with manuscript editing. The authors take responsibility for the final text, analyses, and claims.

ETHICS STATEMENT

This work studies post-training methods for LM agents on public software-engineering and mathematical-reasoning benchmarks. More reliable coding agents could support software maintenance, but may also produce incorrect or insecure patches and facilitate malicious software modification. Our experiments evaluate performance in benchmark environments and do not establish safety for autonomous deployment. Practical use should therefore include sandboxed execution, restricted tool access, rigorous testing, and human review of generated code. Broader societal benefits and risks are discussed in Appendix C.

REPRODUCIBILITY STATEMENT

We specify the rollout protocols and training objectives in Section 3, with the unified formulation derived in Appendix A. Dataset splits, training configurations, and the OpenHands prompts and tool specifications are documented in Appendices D, E, and G, respectively. Appendix F reports the settings and results of the supplementary experiments, including evaluation across multiple sampling seeds.

REFERENCES

Rishabh Agarwal, Nino Vieillard, Yongchao Zhou, Piotr Stanczyk, Sabela Ramos Garea, Matthieu Geist, and Olivier Bachem. On-policy distillation of language models: Learning from self-generated mistakes. In *The twelfth international conference on learning representations*, 2024.

Elad Ben Avraham, Changhao Li, Ron Dorfman, Roy Ganz, Oren Nuriel, Amir Dudai, Aviad Aberdam, Noah Flynn, Elman Mansimov, Adi Kalyanpur, et al. Dream: Deep research evaluation with agentic metrics. *arXiv preprint arXiv:2602.18940*, 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. *arXiv preprint arXiv:2505.20411*, 2025.

Ruisheng Cao, Mouxiang Chen, Jiawei Chen, Zeyu Cui, Yunlong Feng, Binyuan Hui, Yuheng Jing, Kaixin Li, Mingze Li, Junyang Lin, et al. Qwen3-coder-next technical report. *arXiv preprint arXiv:2603.00729*, 2026.

Shiyi Cao, Dacheng Li, Fangzhou Zhao, Shuo Yuan, Sumanth R Hegde, Connor Chen, Charlie Ruan, Tyler Griggs, Shu Liu, Eric Tang, et al. Skyrl-agent: Efficient rl training for multi-turn llm agent. *arXiv preprint arXiv:2511.16108*, 2025.

Jiefeng Chen, Bhavana Dalvi Mishra, Jaehyun Nam, Rui Meng, Tomas Pfister, and Jinsung Yoon. Mars: Modular agent with reflective search for automated ai research. *arXiv preprint arXiv:2602.02660*, 2026.

Mingyang Chen, Haoze Sun, Tianpeng Li, Fan Yang, Hao Liang, Keer Lu, Bin Cui, Wentao Zhang, Zenan Zhou, and Weipeng Chen. Facilitating multi-turn function calling for llms via compositional instruction tuning. *arXiv preprint arXiv:2410.12952*, 2024.

Zijian Chen, Xueguang Ma, Shengyao Zhuang, Ping Nie, Kai Zou, Andrew Liu, Joshua Green, Kshama Patel, Ruoxi Meng, Mingyi Su, et al. Browsecomp-plus: A more fair and transparent evaluation benchmark of deep-research agent. *arXiv preprint arXiv:2508.06600*, 2025.

Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. Introducing SWE-bench Verified. OpenAI Blog, August 2024. URL https://openai.com/index/introducing-swe-bench-verified/.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, et al. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025.

Jiazhan Feng, Shijue Huang, Xingwei Qu, Ge Zhang, Yujia Qin, Baoquan Zhong, Chengquan Jiang, Jinxin Chi, and Wanjun Zhong. Retool: Reinforcement learning for strategic tool use in llms. *arXiv preprint arXiv:2504.11536*, 2025.

Google DeepMind. Gemini 3.1 Pro model card. https://deepmind.google/models/model-cards/gemini-3-1-pro/, February 2026.

Yuxian Gu, Li Dong, Furu Wei, and Minlie Huang. MiniLLM: Knowledge distillation of large language models. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=5h0qf7IBZZ.

Anisha Gunjal, Anthony Wang, Elaine Lau, Vaskar Nath, Yunzhong He, Bing Liu, and Sean Hendryx. Rubrics as rewards: Reinforcement learning beyond verifiable domains. *arXiv preprint arXiv:2507.17746*, 2025.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, YK Li, et al. Deepseek-coder: when the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.

Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8):1–79, 2024.

Jiawei Huang, Qingping Yang, Renjie Zheng, and Jiaze Chen. Beyond verifiable rewards: Rubric-based grm for reinforced fine-tuning swe agents. *arXiv preprint arXiv:2604.16335*, 2026.

Jonas Hübotter, Frederike Lübeck, Lejs Behric, Anton Baumann, Marco Bagatella, Daniel Marta, Ido Hakimi, Idan Shenfeld, Thomas Kleine Buening, Carlos Guestrin, et al. Reinforcement learning via self-distillation. *arXiv preprint arXiv:2601.20802*, 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? *arXiv preprint arXiv:2310.06770*, 2023.

Niklas Lauffer, Xiang Deng, Srivatsa Kundurthy, Brad Kenstler, and Jeff Da. Imitation learning for multi-turn lm agents via on-policy expert corrections. *arXiv preprint arXiv:2512.14895*, 2025.

Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. Large language model-based agents for software engineering: A survey. *arXiv preprint arXiv:2409.02977*, 2024.

Kevin Lu and Thinking Machines Lab. On-policy distillation. *Thinking Machines Lab: Connectionism*, 2025. doi: 10.64434/tml.20251026. https://thinkingmachines.ai/blog/on-policy-distillation.

OpenAI. GPT-5.6 system card. https://deploymentsafety.openai.com/gpt-5-6, July 2026.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with swe-gym. *arXiv preprint arXiv:2412.21139*, 2024.

Rushi Qiang, Yuchen Zhuang, Yinghao Li, Dingu Sagar V K, Rongzhi Zhang, ChangHao Li, Ian Wong, Sherry Yang, Percy Liang, Chao Zhang, and Bo Dai. MLE-Dojo: Interactive environments for empowering LLM agents in machine learning engineering. In *Advances in Neural Information Processing Systems*, volume 38, 2025a.

Rushi Qiang, Yuchen Zhuang, Anikait Singh, Percy Liang, Chao Zhang, Sherry Yang, and Bo Dai. Mle-smith: Scaling mle tasks with automated multi-agent pipeline. *arXiv preprint arXiv:2510.07307*, 2025b.

Stéphane Ross and Drew Bagnell. Efficient reductions for imitation learning. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 661–668. JMLR Workshop and Conference Proceedings, 2010.

Stephane Ross and J Andrew Bagnell. Reinforcement and imitation learning via interactive no-regret learning. *arXiv preprint arXiv:1406.5979*, 2014.

Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pp. 627–635. JMLR Workshop and Conference Proceedings, 2011.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

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.

Haoran Wang, Zhenyu Hou, Yao Wei, Jie Tang, and Yuxiao Dong. Swe-dev: Building software engineering agents with training and inference scaling. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 3742–3761, 2025.

Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*, 50(4):911–936, 2024a.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024b.

Jason Wei, Zhiqing Sun, Spencer Papay, Scott McKinney, Jeffrey Han, Isa Fulford, Hyung Won Chung, Alex Tachard Passos, William Fedus, and Amelia Glaese. Browsecomp: A simple yet challenging benchmark for browsing agents. *arXiv preprint arXiv:2504.12516*, 2025a.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I Wang. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution. *arXiv preprint arXiv:2502.18449*, 2025b.

Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 1482–1494. IEEE, 2023.

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*, 2025a.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

John Yang, Kilian Lieret, Carlos E Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents. *arXiv preprint arXiv:2504.21798*, 2025b.

Sherry Yang, Joy He-Yueya, and Percy Liang. Reinforcement learning for machine learning engineering agents. *arXiv preprint arXiv:2509.01684*, 2025c.

Shihui Yang, Chengfeng Dou, Peidong Guo, Kai Lu, Qiang Ju, Fei Deng, and Rihui Xin. Dcpo: Dynamic clipping policy optimization. *arXiv preprint arXiv:2509.02333*, 2025d.

Tianzhu Ye, Li Dong, Zewen Chi, Xun Wu, Shaohan Huang, and Furu Wei. Black-box on-policy distillation of large language models. *arXiv preprint arXiv:2511.10643*, 2025.

Burak Yetiştiren, Işık Özsoy, Miray Ayerdem, and Eray Tüzün. Evaluating the code quality of ai-assisted code generation tools: An empirical study on github copilot, amazon codewhisperer, and chatgpt. *arXiv preprint arXiv:2304.10778*, 2023.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. Dapo: An open-source llm reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, et al. Multi-swe-bench: A multilingual benchmark for issue resolving. *arXiv preprint arXiv:2504.02605*, 2025.

Hao Zhang, Mingjie Liu, Shaokun Zhang, Songyang Han, Jian Hu, Zhenghui Jin, Yuchi Zhang, Shizhe Diao, Ximing Lu, Binfeng Xu, et al. Prorl agent: Rollout-as-a-service for rl training of multi-turn llm agents. *arXiv preprint arXiv:2603.18815*, 2026.

Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. *arXiv preprint arXiv:2401.07339*, 2024.

Siyan Zhao, Zhihui Xie, Mengchen Liu, Jing Huang, Guan Pang, Feiyu Chen, and Aditya Grover. Self-distilled reasoner: On-policy self-distillation for large language models. *arXiv preprint arXiv:2601.18734*, 2026.

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

A Derivation of the Unified Post-Training View

In this section, we provide additional details on the unified formulation in Section 3.3. The goal is not to claim that all post-training algorithms are identical, but rather to expose a common structure shared by imitation learning, reinforcement learning, on-policy distillation, and our DAgger-style methods. Each method can be viewed as repeatedly constructing a weighted supervised dataset over agent states and turn-level actions, and then updating the policy to increase the likelihood of selected actions under those states.

General form. Consider the i-th post-training iteration. A rollout or data-collection procedure first induces a distribution over contexts, which we denote by p_s. Here a context s corresponds to the full interaction prefix available to the agent before producing the next turn-level action, including previous messages, tool calls, and environment observations. Given a context s, the algorithm then specifies a label distribution $p_a(\cdot \mid s)$ from which the action label a is drawn. Finally, each sampled pair (s, a) may be assigned a scalar score or weight $w(s, a)$, such as an advantage estimate, a distillation coefficient, or simply a constant weight.

This leads to the following population-level update:

$$\theta_{i+1} = \arg\max_{\theta} \; \mathbb{E}_{s\sim \mathrm{sg}(p_s),\, a\sim \mathrm{sg}(p_a(\cdot|s))} \left[\mathrm{sg}(w(s,a)) \log \pi_\theta(a \mid s)\right] - \lambda \Omega_i(\theta), \tag{16}$$

where $\mathrm{sg}(\cdot)$ denotes stop-gradient. The stop-gradient notation emphasizes that the contexts, labels, and weights are treated as fixed data during the policy update: gradients are taken only through the log-likelihood term $\log \pi_\theta(a \mid s)$ and the optional regularizer $\Omega_i(\theta)$. For autoregressive LM agents, a turn-level action $a = (a_1, \ldots, a_m)$ is a sequence of tokens, and

$$\log \pi_\theta(a \mid s) = \sum_{j=1}^{m} \log \pi_\theta(a_j \mid s, a_{<j}). \tag{17}$$

Thus, standard token-level cross-entropy training is a special case of Eq. (16) with unit weights.

In finite-sample form, this corresponds to collecting a batch

$$\mathcal{D}_i = \{(s_k, a_k, w_k)\}_{k=1}^{N},$$

and optimizing

$$\max_{\theta} \; \frac{1}{N} \sum_{k=1}^{N} w_k \log \pi_\theta(a_k \mid s_k) - \lambda \Omega_i(\theta). \tag{18}$$

Different post-training algorithms differ primarily in how they construct the context distribution p_s, where they obtain action labels p_a, and how they define the weights $w(s, a)$.

SFT / behavior cloning. In supervised fine-tuning, trajectories are generated by an expert or teacher policy π_e. Therefore, both the contexts and labels come from expert-induced trajectories. In the notation of Eq. (16),

$$p_s = d^{\pi_e}, \qquad p_a(\cdot \mid s) = \pi_e(\cdot \mid s), \qquad w(s, a) \equiv 1,$$

where d^{π_e} denotes the context distribution induced by rolling out the expert policy. The resulting objective is

$$\max_{\theta} \; \mathbb{E}_{s\sim d^{\pi_e},\, a\sim \pi_e(\cdot|s)} \left[\log \pi_\theta(a \mid s)\right], \tag{19}$$

which is exactly behavior cloning on expert trajectories. If rejection sampling is used to keep only successful expert trajectories, this can be viewed as changing the empirical expert dataset to the retained subset, or equivalently absorbing a success indicator into the data distribution.

RL / policy-gradient methods. In reinforcement learning with verifiable rewards, the policy is trained on trajectories sampled from the current student policy. Thus, both the contexts and executed actions are induced by the student:

$$p_s = d^{\pi_{\theta_i}}, \qquad p_a(\cdot \mid s) = \pi_{\theta_i}(\cdot \mid s).$$

Policy-gradient methods optimize expected return using the score-function identity:

$$\nabla_\theta J(\theta) = \mathbb{E}_{s\sim d^{\pi_{\theta_i}},\, a\sim \pi_{\theta_i}(\cdot|s)} \left[A_i(s,a) \nabla_\theta \log \pi_\theta(a \mid s)\right], \tag{20}$$

where $A_i(s,a)$ is an advantage estimate computed from rewards or test outcomes. This gradient is induced by the surrogate objective

$$\max_{\theta} \mathbb{E}_{s\sim d^{\pi_{\theta_i}},\, a\sim\pi_{\theta_i}(\cdot|s)} \left[\mathrm{sg}(A_i(s,a)) \log \pi_\theta(a \mid s)\right] - \lambda\Omega_i(\theta). \tag{21}$$

Therefore, RL corresponds to

$$p_s = d^{\pi_{\theta_i}}, \qquad p_a = \pi_{\theta_i}, \qquad w(s,a) = A_i(s,a).$$

For GRPO-style training, $A_i(s,a)$ is typically a group-normalized advantage derived from sampled completions and executable feedback. Clipping, KL penalties, or trust-region constraints can be represented abstractly by the optional regularizer $\Omega_i(\theta)$ or by implementation-specific modifications to the surrogate.

On-policy distillation. On-policy distillation also collects contexts and actions from the current student policy, but replaces scalar task reward with a teacher-based distillation signal. The rollout distribution is therefore

$$p_s = d^{\pi_{\theta_i}}, \qquad p_a(\cdot \mid s) = \pi_{\theta_i}(\cdot \mid s).$$

A common view of OPD is that it encourages the student to align with the teacher on states visited by the student itself. In a reverse-KL or score-function view, one can write the teacher-alignment objective at a fixed context s as

$$-D_{\mathrm{KL}}\left(\pi_\theta(\cdot \mid s) \,\|\, \pi_e(\cdot \mid s)\right) = \mathbb{E}_{a\sim\pi_\theta(\cdot|s)} \left[\log \pi_e(a \mid s) - \log \pi_\theta(a \mid s)\right]. \tag{22}$$

Taking the score-function gradient and evaluating the sampling distribution at the current policy π_{θ_i} gives, up to an additive baseline,

$$\mathbb{E}_{a\sim\pi_{\theta_i}(\cdot|s)} \left[\left(\log \pi_e(a \mid s) - \log \pi_{\theta_i}(a \mid s)\right) \nabla_\theta \log \pi_\theta(a \mid s)\right]. \tag{23}$$

Thus OPD can be written in the unified form with

$$p_s = d^{\pi_{\theta_i}}, \qquad p_a = \pi_{\theta_i}, \qquad w(s,a) = \log \pi_e(a \mid s) - \log \pi_{\theta_i}(a \mid s) = -\log \frac{\pi_{\theta_i}(a \mid s)}{\pi_e(a \mid s)}.$$

This highlights the key difference between OPD and SFT: OPD trains on student-induced contexts, but the action labels are still sampled from the student rather than replaced by teacher actions. Consequently, early in training, OPD may spend much of its supervision budget on low-quality or prematurely failed student trajectories.

DAgger-style rollout. Our DAgger-style method modifies both the context distribution and the per-turn supervision. At iteration i, each turn is executed by the teacher with probability β_i and by the student with probability $1-\beta_i$. This induces a turn-level mixture-policy context distribution, denoted by d_i^{turn}. In the main text and Table 2, we use p_i^{turn} as shorthand for this turn-level mixture rollout protocol and its induced state distribution.

Rather than labeling every visited state with the teacher, the label and weight now follow the policy that executed the turn: a teacher-executed turn is supervised by the teacher action under unit-weight cross-entropy, whereas a student-executed turn is supervised by the student's own executed action under the on-policy-distillation weight. By using b as the executor indicator at state s,

$$p_s = d_i^{\mathrm{turn}}, \qquad p_a(\cdot \mid s) = \begin{cases} \pi_e(\cdot \mid s), & b = 1, \\ \pi_{\theta_i}(\cdot \mid s), & b = 0, \end{cases} \qquad w(s,a) = \begin{cases} 1, & b = 1, \\ -\log \dfrac{\pi_{\theta_i}(a \mid s)}{\pi_e(a \mid s)}, & b = 0. \end{cases}$$

Substituting these choices into the general form of Eq. (16) recovers the executor-matched objective of Eq. (14), while taking the teacher branch ($b \equiv 1$) at every state reduces it to pure cross-entropy on teacher labels. As β_i decays over training, the state distribution gradually shifts from mostly teacher-induced toward more student-induced contexts, and the supervision correspondingly anneals from predominantly cross-entropy toward predominantly reverse-KL, reducing the train-test mismatch that limits pure SFT.

AggreVaTe-style rollout. Our AggreVaTe-style variant uses a trajectory-level mixture rather than independent turn-level mixing. At iteration i, we sample a student-prefix length $\kappa \sim \rho_i$. The student executes the first κ turns, after which the teacher completes the remainder of the trajectory. This

induces a context distribution denoted by d_i^{traj}, or equivalently the trajectory-level rollout protocol p_i^{traj} in Table 2.

As in the DAgger-style variant, the supervision follows the executor, now partitioned by the intervention point κ: the student-executed prefix ($t \le \kappa$, $b = 0$) is supervised under the reverse-KL weight, while the teacher-completed suffix ($t > \kappa$, $b = 1$) is supervised under unit-weight cross-entropy. Hence

$$p_s = d_i^{\mathrm{traj}}, \qquad p_a(\cdot \mid s) = \begin{cases} \pi_e(\cdot \mid s), & b = 1, \\ \pi_\theta(\cdot \mid s), & b = 0, \end{cases} \qquad w(s,a) = \begin{cases} 1, & b = 1, \\ -\log \dfrac{\pi_\theta(a \mid s)}{\pi_e(a \mid s)}, & b = 0, \end{cases}$$

which instantiates the same executor-matched objective (Eq. (14)) under the trajectory-level state distribution. Compared with turn-level DAgger-style mixing, this variant exposes the student to longer contiguous prefixes before the teacher recovers and completes the trajectory, while applying the identical per-turn supervision rule.

Summary. The unified view separates three design choices that are often entangled in post-training algorithms:

where states come from (p_s), where labels come from (p_a), how samples are weighted (w).

SFT uses expert states and expert labels with uniform weights, but suffers from covariate shift because deployment states are induced by the student. RL uses student states and student actions, but relies on sparse or noisy advantage estimates. OPD also uses student states and student actions, but replaces rewards with teacher-based log-probability weights, which still leaves it vulnerable to cold-start failures when early student rollouts are unproductive. Our DAgger-style and AggreVaTe-style methods instead use mixture-induced states with executor-matched supervision: teacher labels under cross-entropy on teacher-executed turns and the student's own actions under reverse-KL distillation on student-executed turns, thus combining on-policy state coverage with dense, trajectory-consistent supervision.

B Limitations

Despite strong empirical results, our study has several limitations. First, our experiments focus on software-engineering agents under the OpenHands scaffold, and further work is needed to validate whether the same gains transfer to other long-horizon agentic domains such as web navigation, data analysis, or scientific computing. Second, our method relies on a stronger teacher policy to provide action labels on visited states; its effectiveness may depend on teacher quality, availability, and inference cost. Finally, our failure analysis suggests that the remaining errors are increasingly dominated by long-context limitations such as context overflow, indicating that improvements in memory, retrieval, or context management may be necessary for further gains.

C Potential Social Impact

Potential Positive Societal Impacts. Our work aims to improve the reliability and sample efficiency of post-training methods for long-horizon LM agents. In software engineering, more capable agents could help developers localize bugs, repair code, maintain open-source projects, and reduce the cost of routine debugging and maintenance. By mitigating covariate shift and improving agent stability, DAgger-style training may also make tool-using systems less brittle under deployment-time interactions, enabling more dependable assistance in complex technical workflows. More broadly, the principle of teacher-interleaved state-distribution correction may benefit other interactive domains where agents must reason, act, and recover over many turns, such as data analysis, web automation, and scientific computing.

Potential Negative Societal Impacts. Improving long-horizon software-engineering agents also introduces potential risks. More capable code-editing agents could be misused to automate harmful software modifications, discover exploitable vulnerabilities, or scale malicious development workflows. Even in benign settings, agents may introduce subtle bugs, insecure code, or incorrect patches if deployed without sufficient review and testing. Because our method improves the ability of smaller

models to perform complex software tasks, it may also lower the barrier to both beneficial and harmful automation. We therefore emphasize that such agents should be deployed with safeguards, including sandboxed execution, restricted tool access, human review for code changes, and rigorous testing before patches are merged or released.

D Dataset and Task Details

SWE-Gym. SWE-Gym (Pan et al., 2024) is a software-engineering agent benchmark and training environment built from real GitHub issues. Each task provides a repository, an issue description, and executable tests, requiring the agent to inspect the codebase, localize the bug or missing functionality, edit source files, and submit a patch. We use SWE-Gym as the primary training corpus for SFT, OPD, and our DAgger-style and AggreVaTe-style methods. For in-domain evaluation, we reserve a fixed set of 100 SWE-Gym instances as SWE-Gym Holdout, and use the remaining $2{,}338$ tasks for training.

SWE-Bench Verified. SWE-Bench Verified (Chowdhury et al., 2024) is a curated subset of SWE-Bench (Jimenez et al., 2023) consisting of real-world GitHub issue resolution tasks with human-validated problem statements and evaluation tests. Each instance requires an agent to modify a checked-out repository so that the generated patch satisfies the issue description and passes the hidden regression tests. We use SWE-Bench Verified as our final out-of-domain evaluation benchmark, reporting task-resolution rate under the OpenHands scaffold. Following standard practice, a task is considered resolved only if the submitted patch passes the corresponding evaluation tests.

E Experimental Details

E.1 Overall Experimental Configuration

Unless otherwise specified, our *DAgger-style* and *AggreVaTe-style* methods use the same optimization and rollout-update configuration. At each training iteration, we collect a fresh mixed-policy rollout batch of 512 task instances and update the student on the resulting teacher-labeled data. We train for 5 rollout-update iterations with a constant learning rate of 3×10^{-6}.

For *DAgger-style* sampling, we initialize the teacher-mixture coefficient at $\beta_1 = 1.0$ and decrease it by 0.2 per iteration, with a lower bound of 0:

$$\beta_i = \max(0, 1.0 - 0.2(i-1)).$$

This schedule keeps early trajectories strongly teacher-guided while gradually increasing the fraction of student-induced states as training progresses.

For *AggreVaTe-style* sampling, we initialize the maximum student-prefix length at 0 and increase it by 20 turns per iteration, capped at 100. At iteration i, we draw the student-prefix length from

$$\rho_i = \mathrm{Unif}\{0, \ldots, \min(100, 20(i-1))\}.$$

After the sampled prefix, the teacher completes the remaining trajectory, providing trajectory-level recovery from student-induced states.

For all methods, including baselines, rollout collection uses temperature 0.7 and top-$p = 0.9$, while evaluation uses greedy decoding. Across training and evaluation, we use a maximum context length of 64K tokens and allow at most 100 environment interactions per trajectory. Unless otherwise specified, all reported results use task-resolution rate as the evaluation metric.

Compute and Wall-Clock Cost. The teacher is served on four A100 GPUs and student rollout and training use four more. Each rollout-update iteration with 512 tasks takes 3–4 hours, so a five-iteration run completes in about 18 hours. Teacher inference dominates the runtime, with costs comparable to those of generating teacher trajectories for SFT or labeling student rollouts for OPD. Mixed-policy rollouts use SGLang's native KV caching without additional cache-management logic.

Data Aggregation. Unlike the original DAgger, we do not aggregate data across iterations: each iteration re-collects trajectories on the same task set and updates the student on the current batch only. Replaying earlier batches is compatible with our method and left for future work.

Table 5: Training configurations for the post-training methods.

Configuration	SFT	Ours / OPD	GRPO
Training data	SWE-Gym	SWE-Gym	SWE-Gym
Trajectory source	Teacher policy	Mixture / student policy	Student policy
Filtering	Passing patches	Valid submissions	Executable feedback
Number of tasks per online batch	–	512	64
Generations per task	1	1	8
Online iterations / passes	–	5	1 pass
Epochs per batch	3	3	–
Batch size	16	16	64 tasks
Learning rate	cosine, $10^{-5} \rightarrow 10^{-6}$	constant 3×10^{-6}	constant 1×10^{-6}
Warmup ratio	0.1	–	–
Number of retained trajectories	684	varies by batch	–
GPUs	4 A100	4 A100	4 A100

E.2 Training Implementation Details

We summarize the training configurations for all post-training methods in Table 5. For all 4B and 8B experiments, the teacher is served on 4 A100 GPUs, and student rollout collection and training use another 4 A100 GPUs.

SFT. For supervised fine-tuning, we follow the training configuration of SWE-Gym (Pan et al., 2024). We first use the teacher model to generate trajectories for all SWE-Gym training tasks, and then apply rejection sampling to retain only trajectories whose final patches pass the executable tests. This results in 684 successful trajectories. We train the student for 3 epochs on this filtered dataset with batch size 16, using a cosine learning-rate schedule with maximum learning rate 1×10^{-5}, minimum learning rate 1×10^{-6}, and warmup ratio 0.1.

DAgger-style, AggreVaTe-style, and OPD. Our *DAgger-style* and *AggreVaTe-style* methods use the same optimization configuration as on-policy distillation (OPD), differing only in how trajectories are collected and supervised. At each online iteration, we sample 512 trajectories using either a mixture policy for our methods or the student policy for OPD. We filter out trajectories that do not produce a valid final patch submission. The remaining trajectories are used for training with teacher supervision: our methods train on teacher-labeled actions on visited states, while OPD uses the teacher distillation signal on student-policy trajectories. We use a constant learning rate of 3×10^{-6}, batch size 16, and train for 3 epochs over each collected online batch. In total, we perform 5 online rollout-update iterations.

GRPO. We train GRPO on the full SWE-Gym training split of 2,338 tasks, using the same student initializations as the other post-training methods. Each update uses 64 tasks with 8 trajectories per task, yielding 512 rollouts. We train for one pass over the training split with a constant learning rate of 1×10^{-6}, using executable test outcomes as terminal rewards for policy-gradient updates (Cao et al., 2025).

F Additional Experiments and Ablation Studies

This section presents the setup, results, and analysis of the supplementary experiments indexed in Section 4. We examine rollout schedules, teacher and domain generalization, loss design, evaluation seeds, and alternative divergence measures.

F.1 Sensitivity to the Rollout Schedule

Setup. We use Qwen3-4B and five rollout-update iterations to study how rapidly training should move from teacher-guided to student-induced states. DAgger-style rollouts decrease β_i, the probability of executing a teacher action at each turn. AggreVaTe-style rollouts sample a student-prefix length $\kappa \sim \rho_i = \mathrm{Unif}\{0, \ldots, M_i\}$, after which the teacher completes the trajectory; increasing M_i allows longer student-controlled prefixes, up to the 100-turn horizon. For each protocol, we compare *slow*, *base*, and *fast* schedules (Table 6). The reference baselines are iterative SFT, which collects teacher-only trajectories at each iteration ($\beta_i \equiv 1$), and OPD, which uses student-only trajectories ($\beta_i \equiv 0$) with a reverse-KL objective.

We also evaluate a staged SFT-to-OPD baseline (Lu & Thinking Machines Lab, 2025) to test whether a separate SFT warm start suffices. We initialize OPD from the 4.5K-effective-sample SFT checkpoint in Figure 1, obtained after two supervised passes over the SWE-Gym training set. We then run three OPD iterations, each with 512 rollouts and four training passes, for approximately 10.5K effective samples in total, slightly exceeding the budget of the mixture runs.

Table 6: Rollout-schedule sensitivity with Qwen3-4B. Schedule entries show the initial and final teacher-execution probability β_i (DAgger-style) or maximum student-prefix length M_i (AggreVaTe-style), followed by the per-iteration change. The *fast* schedules reach their limits at the third iteration and remain there. Task-resolution rate (%); higher is better.

Rollout	Schedule	SWE-Gym Holdout	SWE-Bench Verified-100
Iterative SFT	$\beta_i \equiv 1$	15.0%	23.0%
OPD	$\beta_i \equiv 0$	16.0%	24.0%
SFT-to-OPD	–	16.0%	26.0%
DAgger-style	slow ($1.0 \to 0.5$, -0.1)	17.0%	27.0%
	base ($1.0 \to 0$, -0.2)	**19.0%**	29.0%
	fast ($1.0 \to 0$, -0.5)	17.0%	**30.0%**
AggreVaTe-style	slow ($0 \to 50$, $+10$)	15.0%	25.0%
	base ($0 \to 100$, $+20$)	16.0%	26.0%
	fast ($0 \to 100$, $+50$)	**17.0%**	**29.0%**

Results and Analysis. On SWE-Bench Verified-100, all six mixture schedules outperform iterative SFT (23.0%) and OPD (24.0%). Moving from *slow* to *fast* annealing raises task resolution from 27.0% to 30.0% for DAgger-style and from 25.0% to 29.0% for AggreVaTe-style training. Faster annealing also improves AggreVaTe-style training on SWE-Gym Holdout (15.0% to 17.0%), whereas DAgger-style peaks with the *base* schedule (19.0%). These patterns are consistent with teacher guidance providing an early warm start and increased student-state coverage benefiting later training, with the best transition rate depending on the rollout protocol and benchmark.

Despite its larger effective-sample budget, SFT-to-OPD reaches 16.0% on SWE-Gym Holdout and 26.0% on SWE-Bench Verified-100, compared with 17.0%/30.0% for DAgger-style (*fast*) and 17.0%/29.0% for AggreVaTe-style (*fast*). Thus, a separate SFT warm start improves OPD on Verified-100 but does not fully recover the gains from mixed-policy training.

F.2 Generalization Across Teachers and Domains

We examine whether the benefits observed in the main experiments extend to a stronger teacher and to a different domain: multi-turn, tool-integrated mathematical reasoning.

F.2.1 Teacher Strength

Setup. We evaluate DAgger-style training with Qwen3-Coder-30B-A3B-Instruct and Qwen3-Coder-480B-A35B-Instruct as teachers, keeping Qwen3-4B-Instruct-2507 as the student. Both settings use sampled teacher actions for supervision, without requiring teacher logits.

Table 7: Effect of teacher choice on DAgger-style training with a 4B student. Task-resolution rate (%); higher is better.

Method	SWE-Gym Holdout	SWE-Bench Verified-100
Qwen3-4B	5.0%	13.0%
+ DAgger-style (30B teacher)	17.0%	27.0%
+ DAgger-style (480B teacher)	**19.0%**	**30.0%**

Results and Analysis. With the 480B teacher, DAgger-style training reaches 19.0% on SWE-Gym Holdout and 30.0% on SWE-Bench Verified-100, compared with 17.0% and 27.0% with the 30B teacher (Table 7). These gains suggest that the student can benefit from stronger supervision even when the teacher provides only sampled actions. The moderate gains despite the increase in teacher scale may reflect the limited capacity of the 4B student.

F.2.2 Tool-Integrated Mathematical Reasoning

Setup. We use ReTool (Feng et al., 2025) to interleave natural-language reasoning with code execution, training on DAPO-Math-17K (Yu et al., 2025) and evaluating on GSM8K and MATH-500. The teacher is Qwen3-30B-A3B-Instruct-2507 and the student is Qwen3-4B-Instruct-2507. All methods collect $4{,}096$ trajectories per iteration, with a group size of 8 for GRPO.

Table 8: Generalization to tool-integrated mathematical reasoning with a 4B student and a 30B teacher. Accuracy (%); higher is better.

Method	GSM8K	MATH-500
Qwen3-4B	27.4%	23.6%
+ GRPO	52.9%	26.0%
+ Iterative SFT	67.6%	40.6%
+ OPD	62.3%	37.8%
+ Ours (*AggreVaTe-style*)	69.3%	41.2%
+ Ours (*DAgger-style*)	**70.4%**	**42.4%**

Results and Analysis. DAgger-style training achieves 70.4% on GSM8K and 42.4% on MATH-500, improving over the strongest baseline, iterative SFT, by 2.8 and 1.8 percentage points, respectively (Table 8). AggreVaTe-style training also surpasses all baselines, reaching 69.3% and 41.2%. Both rollout protocols thus retain their advantage beyond software engineering.

GRPO improves the base model from 27.4% to 52.9% on GSM8K, but only from 23.6% to 26.0% on MATH-500. This contrast is consistent with more reliable tool invocation benefiting simpler tasks while being insufficient for harder reasoning. All teacher-based methods outperform GRPO on both benchmarks, supporting the value of dense supervision in this setting. Iterative SFT also outperforms OPD, suggesting that high-quality teacher trajectories are particularly useful when early student rollouts provide limited coverage of productive reasoning paths. Teacher-interleaved rollouts combine this guidance with exposure to student-induced states, yielding the strongest results across both benchmarks.

F.3 Effect of Loss Design

Setup. We isolate the supervision signal by fixing the mixture-policy rollout and comparing pure cross-entropy training (Eq. (12)) with executor-matched supervision (Eq. (14)) for both the 4B and 8B students. The latter uses cross-entropy on teacher-executed turns and reverse-KL distillation on student-executed turns. We evaluate both DAgger-style and AggreVaTe-style rollouts on SWE-Gym Holdout and SWE-Bench Verified-100.

Table 9: Effect of loss design: pure cross-entropy vs. executor-matched supervision under identical mixture rollouts. Task-resolution rate (%); higher is better.

Model	Sampling	Loss Design	SWE-Gym Holdout	SWE-Bench Verified-100
Qwen3-4B	DAgger-style	Pure CE	17.0%	26.0%
		CE + OPD	**19.0%**	**26.0%**
	AggreVaTe-style	Pure CE	16.0%	24.0%
		CE + OPD	**19.0%**	**27.0%**
Qwen3-8B	DAgger-style	Pure CE	19.0%	**30.0%**
		CE + OPD	**20.0%**	29.0%
	AggreVaTe-style	Pure CE	17.0%	27.0%
		CE + OPD	**18.0%**	**31.0%**

Results and Analysis. Table 9 shows that executor-matched supervision improves SWE-Gym Holdout in all four settings ($+1.0$ to $+3.0$ points). On SWE-Bench Verified-100, both AggreVaTe-style variants benefit, improving by $+3.0$ and $+4.0$ points at 4B and 8B, respectively. By contrast, DAgger-style training is neutral at 4B and slightly degrades at 8B ($30.0\% \rightarrow 29.0\%$). These results indicate that the supervision rule can complement mixture rollouts, but its benefit depends on the rollout protocol and evaluation benchmark.

F.4 Variance across Evaluation Seeds

Setup. We complement the greedy results in Table 3 with stochastic evaluation of the same 4B base model and post-training checkpoints, using three decoding seeds (0, 21, 42) at temperature 0.7. Each checkpoint remains fixed across the three runs. We report the mean and standard deviation of task-resolution rates; SWE-Bench Verified-100 uses the same subset as Figure 1.

Table 10: Three-seed evaluation of the 4B student (seeds 0, 21, 42; temperature 0.7): mean $\pm$ standard deviation of task-resolution rate (%); higher is better.

Method	SWE-Gym Holdout	SWE-Bench Verified-100	SWE-Bench Verified
Qwen3-4B	$4.0 \pm 0.0\%$	$14.3 \pm 1.5\%$	$11.9 \pm 0.9\%$
+ GRPO	$9.3 \pm 1.5\%$	$16.3 \pm 1.2\%$	$15.9 \pm 1.2\%$
+ SFT	$14.7 \pm 0.6\%$	$23.7 \pm 0.6\%$	$23.2 \pm 0.7\%$
+ OPD	$15.7 \pm 0.6\%$	$24.0 \pm 1.0\%$	$23.6 \pm 0.4\%$
+ Ours (*AggreVaTe-style*)	$15.7 \pm 2.5\%$	$25.3 \pm 1.5\%$	$24.8 \pm 0.4\%$
+ Ours (*DAgger-style*)	$\mathbf{16.7 \pm 0.6\%}$	$\mathbf{29.0 \pm 1.7\%}$	$\mathbf{27.5 \pm 0.3\%}$

Results and Analysis. DAgger-style training achieves the highest mean task-resolution rate across all three evaluation sets (Table 10). On the full SWE-Bench Verified evaluation set, it reaches $27.5 \pm 0.3\%$, exceeding AggreVaTe-style training by 2.7 percentage points and OPD by 3.9 points, with limited variation across decoding seeds. GRPO improves over the base model on all three sets but remains below the teacher-based methods. The mean performance ordering agrees with the greedy evaluation, and the full-set results confirm that the gains extend beyond the Verified-100 subset.

F.5 Additional Divergence Measures

Setup. We complement the reverse-KL analysis in Section 4.4 with forward KL and the symmetric Jensen–Shannon divergence JSD(0.5). We evaluate the 4B student at 3K, 6K, and 9K effective samples on the same student-induced rollouts. Lower values indicate smaller teacher–student divergence.

Table 11: Teacher–student divergence on student-induced rollouts for the 4B student at 3K, 6K, and 9K effective samples; lower is better.

Divergence	Method	3K	6K	9K
Forward KL	SFT	0.082	0.078	0.085
	OPD	0.096	0.085	0.078
	Ours (*DAgger-style*)	**0.080**	**0.077**	**0.072**
JSD(0.5)	SFT	**0.045**	0.043	0.048
	OPD	0.066	0.049	0.043
	Ours (*DAgger-style*)	0.046	**0.041**	**0.039**

Results and Analysis. DAgger-style training achieves the lowest forward KL at all three budgets and the lowest JSD at 6K and 9K (Table 11). At 3K, its JSD is comparable to SFT (0.046 versus 0.045). SFT's divergence decreases initially but rebounds at 9K under both metrics, whereas DAgger-style training continues to improve, reaching 0.072 forward KL and 0.039 JSD. OPD also improves but remains above DAgger-style training at every budget. These trends are consistent with teacher guidance supporting productive early rollouts and increasing student participation maintaining teacher alignment on student-induced states as training scales.

The three metrics capture complementary aspects of teacher–student agreement. Reverse KL, used in the main analysis, weights actions by their probability under the student and assesses whether likely deployment-time behavior is supported by the teacher. Forward KL emphasizes coverage of teacher actions, while JSD treats the two policies symmetrically. The agreement across these measures supports the main finding that teacher-interleaved training reduces policy discrepancy on student-induced states, with the observed advantage extending beyond reverse KL.

G Prompt Templates

G.1 OpenHands System Prompt

We use the OpenHands SWE-agent scaffold throughout training and evaluation. The following system prompt defines the agent role, tool-use behavior, file-system constraints, code-editing policy, version-control rules, and problem-solving workflow.

OpenHands System Prompt

```
You are OpenHands agent, a helpful AI assistant that can interact with a computer to solve
tasks.
<ROLE>
Your primary role is to assist users by executing commands, modifying code, and solving
technical problems effectively.  You should be thorough, methodical, and prioritize
quality over speed.
* If the user asks a question, like "why is X happening", don't try to fix the problem.
Just give an answer to the question.
</ROLE>
<EFFICIENCY>
* Each action you take is somewhat expensive.  Wherever possible, combine multiple actions
into a single action, e.g., combine multiple bash commands into one, using sed and grep to
edit/view multiple files at once.
* When exploring the codebase, use efficient tools like find, grep, and git commands with
appropriate filters to minimize unnecessary operations.
</EFFICIENCY>
<FILE_SYSTEM_GUIDELINES>
* When a user provides a file path, do NOT assume it's relative to the current working
directory.  First explore the file system to locate the file before working on it.
```

```
* If asked to edit a file, edit the file directly, rather than creating a new file with a
different filename.
* For global search-and-replace operations, consider using sed instead of opening file
editors multiple times.
* NEVER create multiple versions of the same file with different suffixes, e.g.,
file_test.py, file_fix.py, file_simple.py.  Instead:
  - Always modify the original file directly when making changes
  - If you need to create a temporary file for testing, delete it once you've confirmed
your solution works
  - If you decide a file you created is no longer useful, delete it instead of creating a
new version
* Do NOT include documentation files explaining your changes in version control unless the
user explicitly requests it.
* When reproducing bugs or implementing fixes, use a single file rather than creating
multiple files with different versions.
</FILE_SYSTEM_GUIDELINES>

<CODE_QUALITY>
* Write clean, efficient code with minimal comments.  Avoid redundancy in comments:  do
not repeat information that can be easily inferred from the code itself.
* When implementing solutions, focus on making the minimal changes needed to solve the
problem.
* Before implementing any changes, first thoroughly understand the codebase through
exploration.
* If you are adding a lot of code to a function or file, consider splitting the function
or file into smaller pieces when appropriate.
* Place all imports at the top of the file unless explicitly requested otherwise or if
placing imports at the top would cause issues, e.g., circular imports, conditional imports,
or imports that need to be delayed for specific reasons.
* If working in a git repo, before you commit code create a .gitignore file if one doesn't
exist.  And if there are existing files that should not be included then update the
.gitignore file as appropriate.
</CODE_QUALITY>

<VERSION_CONTROL>
* If there are existing git user credentials already configured, use them and add
Co-authored-by:  openhands <openhands@all-hands.dev> to any commits messages you make.  If
a git config doesn't exist use "openhands" as the user.name and "openhands@all-hands.dev"
as the user.email by default, unless explicitly instructed otherwise.
* Exercise caution with git operations.  Do NOT make potentially dangerous changes, e.g.,
pushing to main, deleting repositories, unless explicitly asked to do so.
* When committing changes, use git status to see all modified files, and stage all files
necessary for the commit.  Use git commit -a whenever possible.
* Do NOT commit files that typically shouldn't go into version control, e.g.,
node_modules/, .env files, build directories, cache files, large binaries, unless
explicitly instructed by the user.
* If unsure about committing certain files, check for the presence of .gitignore files or
ask the user for clarification.
* When running git commands that may produce paged output, e.g., git diff, git log, git
show, use git -no-pager <command> or set GIT_PAGER=cat to avoid output being stuck in an
interactive pager.
</VERSION_CONTROL>

<PULL_REQUESTS>
* Important:  Do not push to the remote branch and/or start a pull request unless
explicitly asked to do so.
* When creating pull requests, create only ONE per session/issue unless explicitly
instructed otherwise.
* When working with an existing PR, update it with new commits rather than creating
additional PRs for the same issue.
* When updating a PR, preserve the original PR title and purpose, updating description
only when necessary.
</PULL_REQUESTS>

<PROBLEM_SOLVING_WORKFLOW>
1.  EXPLORATION: Thoroughly explore relevant files and understand the context before
proposing solutions.
2.  ANALYSIS: Consider multiple approaches and select the most promising one.
3.  TESTING:
  * For bug fixes:  Create tests to verify issues before implementing fixes.
  * For new features:  Consider test-driven development when appropriate.
  * Do NOT write tests for documentation changes, README updates, configuration files, or
other non-functionality changes.
  * If the repository lacks testing infrastructure and implementing tests would require
extensive setup, consult with the user before investing time in building testing
infrastructure.
  * If the environment is not set up to run tests, consult with the user first before
investing time to install all dependencies.
  * Do not use mocks in tests unless strictly necessary and justify their use when they
are used.  You must always test real code paths in tests, NOT mocks.
4.  IMPLEMENTATION:
```

```
  * Make focused, minimal changes to address the problem.
  * Always modify existing files directly rather than creating new versions with different
suffixes.
  * If you create temporary files for testing, delete them after confirming your solution
works.
5.  VERIFICATION: If the environment is set up to run tests, test your implementation
thoroughly, including edge cases.  If the environment is not set up to run tests, consult
with the user first before investing time to run tests.
</PROBLEM_SOLVING_WORKFLOW>

<SECURITY>
* Only use GITHUB_TOKEN and other credentials in ways the user has explicitly requested
and would expect.
* Use APIs to work with GitHub or other platforms, unless the user asks otherwise or your
task requires browsing.
</SECURITY>

<EXTERNAL_SERVICES>
* When interacting with external services like GitHub, GitLab, or Bitbucket, use their
respective APIs instead of browser-based interactions whenever possible.
* Only resort to browser-based interactions with these services if specifically requested
by the user or if the required operation cannot be performed via API.
</EXTERNAL_SERVICES>

<ENVIRONMENT_SETUP>
* When user asks you to run an application, don't stop if the application is not installed.
Instead, please install the application and run the command again.
* If you encounter missing dependencies:
  1.  First, look around in the repository for existing dependency files, e.g.,
requirements.txt, pyproject.toml, package.json, Gemfile, etc.
  2.  If dependency files exist, use them to install all dependencies at once, e.g., pip
install -r requirements.txt, npm install, etc.
  3.  Only install individual packages directly if no dependency files are found or if
only specific packages are needed.
* Similarly, if you encounter missing dependencies for essential tools requested by the
user, install them when possible.
</ENVIRONMENT_SETUP>

<TROUBLESHOOTING>
* If you've made repeated attempts to solve a problem but tests still fail or the user
reports it's still broken:
  1.  Step back and reflect on 5-7 different possible sources of the problem.
  2.  Assess the likelihood of each possible cause.
  3.  Methodically address the most likely causes, starting with the highest probability.
  4.  Explain your reasoning process in your response to the user.
* When you run into any major issue while executing a plan from the user, please don't try
to directly work around it.  Instead, propose a new plan and confirm with the user before
proceeding.
</TROUBLESHOOTING>

<PROCESS_MANAGEMENT>
* When terminating processes:
  - Do NOT use general keywords with commands like pkill -f server or pkill -f python as
this might accidentally kill other important servers or processes.
  - Always use specific keywords that uniquely identify the target process.
  - Prefer using ps aux to find the exact process ID (PID) first, then kill that specific
PID.
  - When possible, use more targeted approaches like finding the PID from a pidfile or
using application-specific shutdown commands.
</PROCESS_MANAGEMENT>
```

G.2 OpenHands Tool Specifications

In addition to the system prompt, the OpenHands scaffold defines a small set of tools that form the agent's action space. We use the following tool specifications throughout trajectory collection and evaluation.

OpenHands Tool Specifications

Tool 1: **execute_bash**
Type: function
Name: execute_bash
Description: Execute a bash command in the terminal.

```
* The command is executed in a bash shell with bash -c.
* It times out after 120 seconds if no output is produced.
* For long-running commands, e.g., servers, the agent should run them in the background
using & or redirect output.
```

* Interactive commands that require user input should be avoided.
* Command output may be truncated if it exceeds the maximum length.
* The agent should use cd to navigate to the correct directory before running commands.

Parameters:
  command: string. The bash command to execute.
Required: command.

Tool 2: str_replace_editor
Type: function
Name: str_replace_editor
Description: A file editor tool that allows viewing and editing files.
Supported commands:
  view: display file contents.
  create: create a new file.
  str_replace: replace an exact string in a file.
  insert: insert text after a specified line.
Critical requirements for str_replace:
* The old_str parameter must match exactly one occurrence in the file.
* The agent should include enough context in old_str to make it unique.
* The new_str parameter contains the replacement text.
* When viewing a directory path, the tool lists non-hidden files up to 2 levels deep.

Parameters:
  command: string, one of view, create, str_replace, or insert. The editor command to run.
  path: string. Absolute path to the file.
  file_text: string. Required for create; the content of the file to create.
  old_str: string. Required for str_replace; the exact string to replace.
  new_str: string. Required for str_replace; the replacement string. For insert, the string to insert.
  insert_line: integer. Required for insert; new_str is inserted after this line.
  view_range: array of integers. Optional for view; specifies [start_line, end_line].
Required: command, path.

Tool 3: finish
Type: function
Name: finish
Description: Signal that the task has been completed. The agent uses this tool when it believes the problem is solved and all necessary changes have been made.
Parameters:
  message: string. A brief summary of what was done to solve the task. Defaults to an empty string.
Required: message.

G.3 Initial User Message

For each SWE task, we construct the initial user message from the issue description and optional hints provided by the dataset. The message instructs the agent to work inside the repository at `/testbed`, avoid modifying tests, and make minimal changes to non-test files.

Initial User Message

I have access to a python code repository in the directory /testbed . You can explore and modify files using the available tools. Consider the following issue description:

<issue_description>
{problem_statement}
</issue_description>

<hints>
{hints_text}
</hints>

Can you help me implement the necessary changes to the repository so that the requirements specified in the <issue_description> are met?
I have already taken care of all changes to any of the test files described in the <issue_description>. This means you DON'T have to modify the testing logic or any of the tests in any way!
Also the development Python environment is already set up for you, i.e., all dependencies already installed, so you don't need to install other packages.
Your task is to make the minimal changes to non-test files in the /testbed directory to ensure the <issue_description> is satisfied.