ANYTOOL-AGENT: API-GUIDED TOOL USE FOR DATA-SCARCE ECOSYSTEMS VIA EXECUTION-GROUNDED POLICY OPTIMIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Tool-using agents can perform well with familiar APIs, but adapting them to a new API ecosystem presents two challenges. First, demonstrations involving unfamiliar APIs are scarce, making it difficult to obtain successful trajectories for an initial policy. With binary outcome rewards, GRPO assigns zero advantages when all rollouts in a group fail, providing no outcome-based learning signal. Second, Vanilla GRPO applies a shared trajectory-level advantage across generated tokens, failing to distinguish necessary, redundant, and erroneous API calls within a multi-step trajectory. API specifications and existing environment entities guide demonstration construction, while sandbox feedback supports action-level credit assignment. We introduce **AnyTool-Agent**, which combines *API-Guided Data Synthesis* with *Execution-Grounded Policy Optimization (EGPO)*. API-Guided Data Synthesis selects API combinations using their specifications and grounds them in existing environment entities to construct tasks bottom-up. Expert trajectories passing semantic completion checks provide supervised fine-tuning demonstrations, while their core API sequences, successfully re-executed in fresh sandboxes, provide verified references for reinforcement learning. EGPO scores actions through reference matching and deterministic execution feedback, with within-group scarcity weighting for infrequently visited reference APIs. These scores multiplicatively reweight the outcome advantage to produce action-level advantages, preserving its sign under positive reweighting factors while distinguishing action contributions. Under the same task-generation budget, API-Guided Data Synthesis covers $2.4\times$ as many distinct APIs as random exploration. On AppWorld and BFCL, EGPO improves avg@4 over Vanilla GRPO by 4.8–11.2 percentage points across two Qwen3 backbones with identical SFT initialization.

1 INTRODUCTION

Tool-using agents can complete multi-step tasks in external environments by interleaving reasoning with API calls (Yao et al., 2023; Schick et al., 2023; Patil et al., 2024; Qin et al., 2024; Du et al., 2024; Li et al., 2025). Given representative demonstrations, supervised fine-tuning (SFT) can equip agents with basic tool-use capabilities, while outcome-based reinforcement learning can further improve task completion (Shao et al., 2024; Qian et al., 2025; Feng et al., 2025a). Adapting such agents to a new API ecosystem, however, remains challenging when API specifications and sandbox access are available but task demonstrations are scarce. For example, a cross-application request-approval task requires the agent to identify the relevant records and execute the correct state-changing operations in sequence. With limited supervision, it may repeatedly inspect available information without completing the requested action.

This setting presents two challenges. First, demonstrations involving unfamiliar APIs are scarce, limiting the initial policy's ability to generate successful trajectories. Manually annotating cross-application tasks is costly because valid solutions depend on environment-specific entities and states. Prior work reduces annotation effort through specification-based task synthesis with solution verification (Liu et al., 2024; Prabhakar et al., 2025; Xu et al., 2025), or through task discovery from exploratory trajectories (Zhai et al., 2025; Wang et al., 2024a). However, top-down synthesis writes queries before solutions exist, so few survive verification on unfamiliar APIs, while exploratory

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

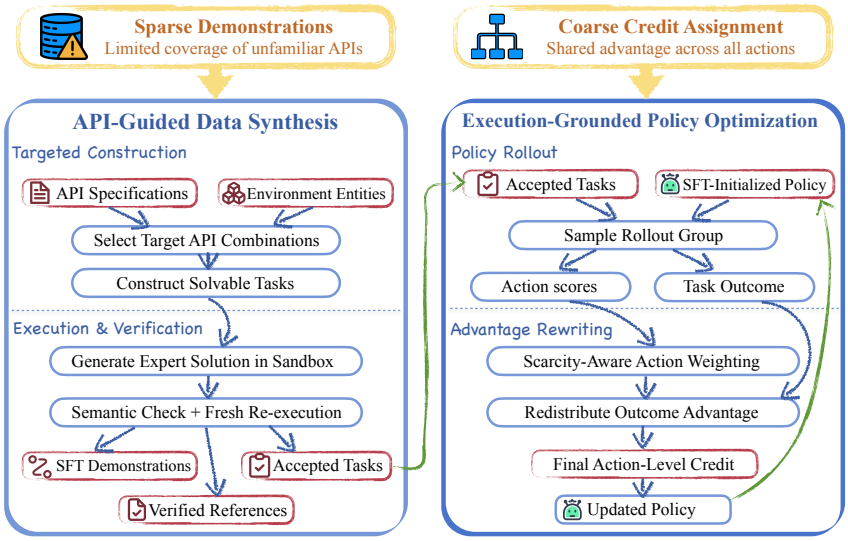


Figure 1: Overview of AnyTool-Agent. **Left**: API-Guided Data Synthesis selects target API combinations from the specifications of a new ecosystem and constructs tasks bottom-up around them. Solutions that pass a semantic completion check and re-execute successfully in a fresh sandbox are retained as reference trajectories. **Right**: Execution-Grounded Policy Optimization (EGPO) rescales the GRPO outcome advantage using sandbox execution and these reference trajectories. Centered action scores multiplicatively reweight the outcome advantage while preserving its sum and sign.

discovery reaches only the compositions its trajectories happen to hit. Neither route grounds task construction in entities already present in the environment. As a result, the initial policy may receive insufficient supervision for tasks that require particular API compositions. Under binary outcome rewards, the policy may then fail on every rollout in a group, yielding zero group-relative advantages and no learning signal for Vanilla GRPO.

Second, Vanilla GRPO provides only coarse-grained credit assignment for multi-step tool use. Specifically, it assigns the same trajectory-level advantage to every generated token and therefore does not distinguish among necessary, redundant, and erroneous API calls (Shao et al., 2024). As a result, useful calls preceding a decisive error may be penalized along with the failed trajectory, while unnecessary calls in a successful trajectory may be reinforced. Process supervision offers a natural remedy by evaluating intermediate steps separately (Lightman et al., 2024). When implemented with learned evaluators, however, it incurs additional inference costs and may produce inaccurate scores (Liu et al., 2025; Ma et al., 2026; Qian et al., 2025). Even with reliable process scores, naively adding them to the outcome advantage can reverse the effective advantage assigned to an action, causing parts of an unsuccessful trajectory to receive positive reinforcement. Action-level feedback should therefore distinguish the contributions of individual API calls without overriding the optimization direction determined by the final outcome.

API specifications and sandbox execution provide complementary signals for these two challenges. First, to address the scarcity of demonstrations for unfamiliar APIs, API specifications guide the construction of demonstrations for target API combinations. Second, deterministic execution feedback provides action-level signals for credit assignment beyond trajectory-level rewards. Accordingly, we introduce **AnyTool-Agent**, which combines *API-Guided Data Synthesis* with *Execution-Grounded Policy Optimization (EGPO)* (Figure 1). The first stage, API-Guided Data Synthesis, departs from prior specification-based synthesis methods (Liu et al., 2024; Prabhakar et al., 2025; Xu et al., 2025), which also read API specifications but generate queries first and then search for solutions, so few survive verification on unfamiliar APIs. Instead, it selects a target API combination and grounds it in existing environment entities, building each task bottom-up so that a solvable trajectory exists by construction. Candidate expert solutions are checked for semantic completion, after which their core API sequences are extracted and re-executed in a fresh sandbox. Accepted expert trajectories provide demonstrations for SFT, while their verified core sequences serve as references during reinforcement learning. The second stage, EGPO, scores each action during policy optimization

using matches to verified reference sequences and deterministic sandbox execution feedback, with within-group scarcity weighting for infrequently visited reference APIs. Rather than adding these scores to the outcome advantage, EGPO mean-centers them within each trajectory and uses them to multiplicatively reweight the advantage across actions. This yields action-level advantages of different magnitudes while preserving the total advantage across the trajectory and, under positive reweighting factors, the sign of the original outcome advantage. API-Guided Data Synthesis increases the likelihood of obtaining rollout groups with informative outcome variation, while EGPO distributes the resulting nonzero advantages more precisely across actions.

We evaluate AnyTool-Agent on AppWorld (Trivedi et al., 2024) and BFCL multi-turn (Patil et al., 2025) using Qwen3-4B-Instruct-2507 and Qwen3-8B-Base. Under the same task-generation budget and verification procedure, API-Guided Data Synthesis covers 325 distinct APIs, compared with 135 under random exploration, exposing the policy to a substantially broader range of unfamiliar APIs during training. Across both backbones and benchmarks, EGPO consistently outperforms the evaluated policy-optimization baselines from the same SFT initialization, improving avg@4 over Vanilla GRPO by up to 11.2 percentage points on AppWorld and 5.7 on BFCL multi-turn. Ablation studies support the contributions of scarcity weighting and multiplicative advantage reweighting, while statistics over sampled trajectories indicate fewer redundant query loops under EGPO.

Main Contributions. (1) We propose **API-Guided Data Synthesis**, which selects target API combinations before task construction. Semantic completion checking and re-execution in a fresh sandbox yield executable SFT demonstrations and verified core API sequences as references for reinforcement learning. (2) We introduce **Execution-Grounded Policy Optimization (EGPO)**, which combines verified reference sequences, deterministic sandbox feedback, and within-group scarcity weighting to estimate action-level credit and multiplicatively redistribute the outcome advantage across actions without changing its total or sign under positive reweighting factors. (3) We conduct extensive experiments on AppWorld and BFCL multi-turn, demonstrating that targeted synthesis broadens API coverage and that EGPO consistently improves task performance. Ablation studies further support the contributions of scarcity weighting and multiplicative advantage reweighting.

2 METHODOLOGY

2.1 FRAMEWORK OVERVIEW

AnyTool-Agent comprises two stages: *API-Guided Data Synthesis* (§2.2) and *Execution-Grounded Policy Optimization (EGPO)* (§2.3), as illustrated in Figure 1. In the first stage, API-Guided Data Synthesis selects target API combinations from API specifications and constructs tasks bottom-up around them. For each task, an expert trajectory is generated in the sandbox and retained as an SFT demonstration if it passes the semantic completion check. The accepted trajectory is then pruned to obtain a core action sequence, which is re-executed in a fresh sandbox. If the re-execution succeeds, the resulting complete trajectory is stored as the reference trajectory τ^*_{GT}. In the second stage, EGPO derives action-level scores from the reference trajectories and deterministic feedback from sandbox execution, and then uses these scores to multiplicatively reweight the GRPO outcome advantage.

2.2 API-GUIDED DATA SYNTHESIS

API-Guided Data Synthesis constructs executable training tasks whose solutions invoke explicitly selected API combinations. The pipeline proceeds in three stages: *targeted task construction*, *execution and semantic verification*, and *reference trajectory construction*. For accepted tasks, it produces two forms of supervision: full expert trajectories for SFT and verified reference trajectories τ^*_{GT} for action-level credit assignment during reinforcement learning.

Targeted Task Construction. API specifications describe the calling conventions and capabilities of individual APIs, allowing target combinations to be selected before task generation rather than discovered through exploration. A selector first samples a target API combination c from the specifications, either within a single application or across multiple applications, and retrieves existing environment entities accessible through the selected APIs. Conditioned on c and these entities, a generator constructs an instruction q. Every object referenced in q exists in the current environment and is described by attributes that the target APIs can resolve, rather than by identifiers that become

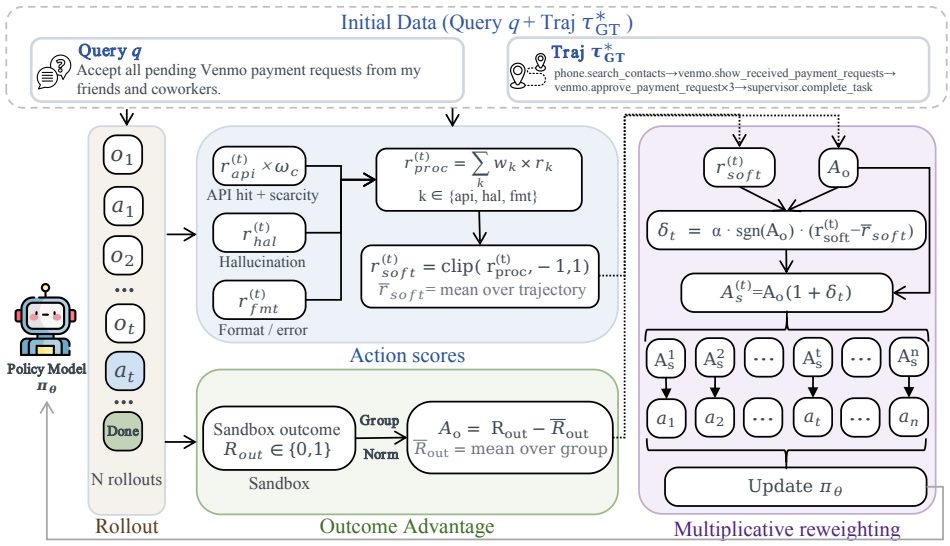


Figure 2: EGPO rescales the outcome advantage. Given a query q, the rollout is compared against the reference trajectory τ^*_{GT} to estimate action-level credit from API hits, hallucination, and format. Centered action scores multiplicatively reweight the outcome advantage. Mean centering preserves the action-level sum, and positive weights preserve its sign.

available only during execution. This bottom-up construction makes API coverage an explicit consequence of the selected combinations rather than of the paths reached through exploration.

Execution and Semantic Verification. A synthesized instruction may be infeasible or incompatible with its target API combination, even when grounded in existing entities. An expert model therefore executes each candidate instruction q in a sandbox to produce a full trajectory. The sandbox checks execution consistency: each API call must operate on the state resulting from preceding calls, and recorded state transitions must agree with returned observations. However, execution consistency does not guarantee task completion. Since synthesized instructions lack task-specific state checkers, a separate LLM verifier assesses the instruction, reference solution, and full trajectory, assigning a semantic verification score $s_{\text{judge}} \in [0, 1]$. We retain a candidate only if it passes the execution checks and satisfies $s_{\text{judge}} \geq 0.85$.

Reference Trajectory Construction. A retained expert trajectory may still contain failed calls or redundant steps that are unnecessary for task completion. We remove failed and repeated calls, bind identifiers to values returned by the APIs, and obtain a core action sequence sufficient to complete the instruction. The sequence is then re-executed in a fresh sandbox. If the re-execution succeeds, the resulting rollout is stored as the verified reference trajectory τ^*_{GT}.

The full expert trajectory and the reference trajectory serve different purposes. SFT uses the full trajectory because it preserves the complete reasoning and interaction trace, whereas τ^*_{GT} retains the core actions needed for credit assignment during reinforcement learning. The corresponding instruction is also added to the reinforcement learning task pool. We analyze how this SFT initialization affects the proportion of all-zero rollout groups in §3.2. The next section presents how EGPO distributes nonzero outcome advantages across actions.

2.3 Execution-Grounded Policy Optimization (EGPO)

Section 2.2 provides executable tasks, their reference trajectories τ^*_{GT}, and an SFT initialization. EGPO uses these references together with deterministic execution feedback to refine action-level credit under sparse terminal supervision. It consists of three components: *execution-grounded action scoring*, *scarcity weighting*, and *multiplicative advantage reweighting*.

Motivation and Formulation. For N rollouts per query, GRPO computes the outcome advantage of rollout b as $A_o^{(b)} = R^{(b)} - \bar{R}$, where $R^{(b)} \in \{0, 1\}$ is the terminal outcome and $\bar{R}$ is the group mean (Shao et al., 2024); we omit the rollout index below. GRPO assigns the same outcome advantage to every generated token and therefore cannot distinguish actions that contribute to task completion from those that do not. A direct approach is to add a centered action-level score, yielding $A_{\text{add}}^{(t)} = A_o + \lambda(r_{\text{soft}}^{(t)} - \bar{r}_{\text{soft}})$, where $r_{\text{soft}}^{(t)} \in [-1, 1]$ and $\bar{r}_{\text{soft}}$ is its within-trajectory mean. This additive formulation introduces a trade-off: a small λ provides little differentiation among actions, whereas a sufficiently large additive term may offset a negative A_o, yielding a positive advantage for an action in a failed rollout. Clipping and centering alone do not rule out such sign reversals (Theorem 3). If these scores are obtained from language-model judges, evaluating individual actions also requires repeated inference and introduces additional scoring noise over long trajectories.

EGPO instead treats action scores derived from verified reference trajectories and deterministic execution feedback as relative credit signals rather than additional rewards. It defines the advantage of action t as $A_s^{(t)} = A_o(1 + \delta^{(t)})$, where $\delta^{(t)}$ is an action-specific reweighting term (Figure 2). The outcome advantage determines the update direction, while $\delta^{(t)}$ adjusts its magnitude using action-level feedback. Provided that $1 + \delta^{(t)} > 0$, $A_s^{(t)}$ retains the sign of A_o; when $A_o = 0$, all action-level advantages remain zero. This formulation requires no learned action-level judge, and the semantic verifier used for data synthesis does not enter the advantage computation.

Execution-grounded Action Scoring. Each action is a code block that issues one or more API calls through the environment's REPL interface (Appendix A.3). Its score aggregates deterministic signals from the execution of these calls. For action t, we define its action score as

$$r_{\text{proc}}^{(t)} = w_{\text{api}} r_{\text{api}}^{(t)} + w_{\text{hal}} r_{\text{hal}}^{(t)} + w_{\text{fmt}} r_{\text{fmt}}^{(t)}. \tag{1}$$

We compare the API calls in each action against the reference API set extracted from τ_{GT}^*. The first occurrence of a reference API receives full match credit. The terminal call `complete_task` invokes environment verification: a successful submission receives positive credit, whereas a premature submission is penalized. The hallucination term r_{hal} penalizes, from execution feedback, calls that do not advance the task. A repeat hit receives discounted credit only when both its call arguments and resulting observation change; otherwise, it receives none. Calls to APIs absent from the toolset and calls that raise runtime errors are penalized directly.

Scarcity Weighting. Because GRPO centers terminal rewards within each rollout group, the outcome advantages satisfy $\sum_{b=1}^{N} A_o^{(b)} = 0$. Updates associated with an API that appears in nearly every rollout therefore tend to cancel between successful and failed rollouts. By contrast, a reference API reached by only a few rollouts provides a more discriminative signal. We accordingly weight reference matches by their within-group scarcity.

For a group of N rollouts, let $h_c^{(b)} \in \{0, 1\}$ indicate whether rollout b invokes API c, and let μ_c and σ_c denote the mean and standard deviation of $\{h_c^{(b)}\}_{b=1}^{N}$. For each API reached by at least one rollout, i.e., $\mu_c > 0$, we define the scarcity multiplier as

$$\omega_c = \text{clip}\left(\frac{1 - \mu_c}{\sigma_c + \epsilon}, 0, \kappa\right). \tag{2}$$

For each API c invoked in action t, its reference-match contribution is multiplied by ω_c before the call-level contributions are aggregated into $r_{\text{api}}^{(t)}$.

Multiplicative Advantage Reweighting. After scarcity weighting modifies the reference-match component of $r_{\text{proc}}^{(t)}$, we clip the resulting action score to $[-1, 1]$, center it within each trajectory, and use it to reweight the outcome advantage:

$$r_{\text{soft}}^{(t)} = \text{clip}\left(r_{\text{proc}}^{(t)}, -1, 1\right),$$

$$\delta^{(t)} = \alpha \ \text{sgn}(A_o) \left(r_{\text{soft}}^{(t)} - \bar{r}_{\text{soft}}\right), \tag{3}$$

$$A_s^{(t)} = A_o \left(1 + \delta^{(t)}\right). \tag{4}$$

Here, $\bar{r}_{\text{soft}} = \frac{1}{T}\sum_{t=1}^{T} r_{\text{soft}}^{(t)}$ is the mean clipped action score within a trajectory of T actions, and α controls the reweighting strength. When $A_o > 0$, actions with above-average scores receive larger positive advantages. When $A_o < 0$, they receive negative advantages of smaller magnitude rather than positive advantages. EGPO couples action-level modulation to the trajectory outcome: centered execution scores, including both rewards and penalties, rescale A_o, so their effect adapts to the magnitude and direction of the outcome advantage rather than acting as an outcome-independent process reward. Provided that $1 + \delta^{(t)} > 0$, every action-level advantage retains the sign of A_o. Theorem 2 gives the corresponding condition on α, and Appendix E.2 verifies that it holds under our configuration. Moreover, because the centered scores sum to zero, $\sum_{t=1}^{T} A_s^{(t)} = TA_o$. We assign $A_s^{(t)}$ to each token belonging to action t. This preservation property therefore applies to the sum of action-level advantages rather than to a token-weighted sum. Substituting these advantages for GRPO's uniform outcome advantage, while retaining its KL regularization, gives

$$\mathcal{J}(\theta) = \mathbb{E}\left[\frac{1}{N}\sum_{b=1}^{N}\frac{1}{L^{(b)}}\sum_{i=1}^{L^{(b)}}\min\Big(\rho_i^{(b)}A_{s,i}^{(b)}, \text{clip}\Big(\rho_i^{(b)}, 1-\varepsilon, 1+\varepsilon\Big)A_{s,i}^{(b)}\Big) - \beta\,\mathbb{D}_{\text{KL}}[\pi_\theta \,\|\, \pi_{\text{ref}}]\right]. \quad (5)$$

where $A_{s,i}^{(b)}$ is the reweighted advantage for the action containing token i, and $\rho_i^{(b)} = \frac{\pi_\theta\left(o_i^{(b)}|q,o_{<i}^{(b)}\right)}{\pi_{\theta_{\text{old}}}\left(o_i^{(b)}|q,o_{<i}^{(b)}\right)}$ is the importance ratio between the current policy and the old policy used to generate rollouts.

3 Experiments

3.1 Experimental Setup

Benchmarks. We evaluate AnyTool-Agent on AppWorld (Trivedi et al., 2024) and the multi-turn tool-use setting of BFCL (Patil et al., 2025). AppWorld tasks typically involve 10–30 API calls, whereas BFCL multi-turn tasks involve 2–5. Both benchmarks provide task queries, public API specifications, and executable environments, but no expert solution trajectories for training. This creates a demonstration-scarce setting. For AppWorld, we construct SFT demonstrations from its official training split through API-Guided Data Synthesis. For BFCL, we synthesize training tasks using its API specifications and executable environment; the official multi-turn evaluation queries and reference solutions are excluded from training. Dataset construction, RL task pools, split usage, and leakage controls are detailed in Appendix B.

Baselines. We conduct experiments with two Qwen3 backbones (Yang et al., 2025): Qwen3-4B-Instruct-2507 and Qwen3-8B-Base. We report zero-shot and SFT results and compare EGPO with Vanilla GRPO, DAPO (Yu et al., 2025), and GSPO (Zheng et al., 2025). For each backbone, all policy-optimization methods are initialized from the same SFT checkpoint and trained on the same RL pool under the same protocol, with a rollout group size of $N = 8$. Vanilla GRPO applies the trajectory-level outcome advantage A_o uniformly to all generated tokens without action-level feedback, while DAPO and GSPO likewise use a single scalar advantage for each trajectory.

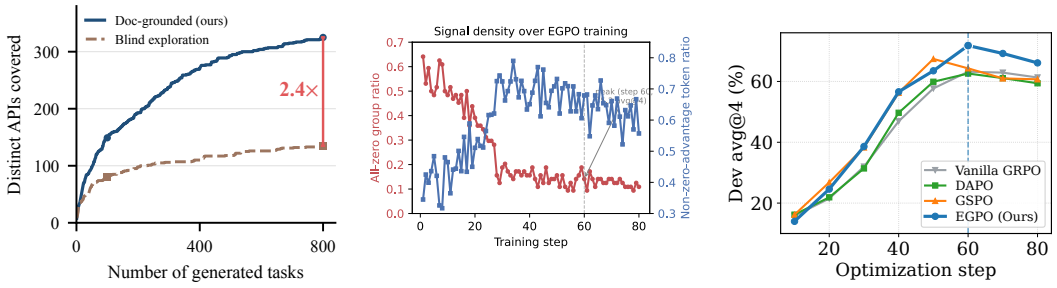
Metrics. We report avg@4 and best@4. For each task, avg@4 averages success over four rollouts, whereas best@4 counts the task as successful if at least one rollout succeeds. On AppWorld, we evaluate on the official dev split every five training steps, using four rollouts per task, and select the checkpoint with the highest dev avg@4.

3.2 Main Results

API-Guided Data Synthesis expands coverage and improves the SFT initialization. Demonstration scarcity has two measurable consequences during training. First, exploration alone reaches only a narrow set of APIs. Second, under outcome-only rewards, most rollout groups fail together. At 800 generated candidates per method, API-Guided Data Synthesis covers 325 distinct APIs, compared with 135 under random exploration (2.4×; Figure 3a). The gap is already evident after 100 generated candidates, with 149 APIs covered by targeted synthesis and 80 by exploration. Both procedures use the same sandbox, grader, and post-filter, differing only in task generation. By selecting target API combinations before constructing tasks, API-Guided Data Synthesis makes coverage an explicit objective; exploration is instead limited to capabilities reached by the generating policy. SFT on the synthesized data reduces, but does not eliminate, all-failure rollout groups.

Method	Model	AppWorld TGC (%) avg@4	AppWorld TGC (%) best@4	BFCL Acc (%) avg@4	BFCL Acc (%) best@4
Proprietary chat models (zero-shot reference)					
Qwen3.7-Max	Qwen3.7-Max	68.5	83.3	56.9	73.0
GLM-5.2	GLM-5.2	62.9	78.6	56.4	72.0
DeepSeek-V4-Pro	DeepSeek-V4-Pro	47.2	72.0	61.5	73.5
Qwen3.5-Plus	Qwen3.5-Plus	49.3	62.5	52.4	69.2
Open-weight chat model (zero-shot reference)					
Qwen3-32B	Qwen3-32B	14.6	26.2	54.2	68.0
AnyTool-Agent (Qwen3-4B-Instruct-2507)					
Zero-shot	Qwen3-4B-Instruct-2507	5.4	11.9	22.3	28.4
+ Data Synthesis + SFT	Qwen3-4B-Instruct-2507	18.5	28.4	40.8	58.9
+ Vanilla GRPO	Qwen3-4B-Instruct-2507	49.7	70.8	54.7	70.2
+ DAPO	Qwen3-4B-Instruct-2507	51.8	75.0	55.7	67.8
+ GSPO	Qwen3-4B-Instruct-2507	52.5	72.0	58.0	70.8
+ EGPO (Ours)	Qwen3-4B-Instruct-2507	**55.2**	**77.4**	**60.4**	**74.0**
AnyTool-Agent (Qwen3-8B-Base)					
Zero-shot	Qwen3-8B-Base	6.5	14.3	38.5	54.0
+ Data Synthesis + SFT	Qwen3-8B-Base	15.5	29.2	52.1	66.3
+ Vanilla GRPO	Qwen3-8B-Base	36.6	64.3	55.8	72.2
+ DAPO	Qwen3-8B-Base	42.0	68.5	59.0	69.8
+ GSPO	Qwen3-8B-Base	42.7	65.5	58.6	72.3
+ EGPO (Ours)	Qwen3-8B-Base	47.8	70.8	60.6	73.0

Table 1: Main results on AppWorld (Task Goal Completion, TGC) and BFCL (multi-turn accuracy).



(a) API coverage (b) Nonzero advantage (c) TGC (avg@4)

Figure 3: **(a)** API-Guided Data Synthesis covers 2.4× as many APIs as random exploration with 800 candidates per method and identical sandbox, grader, and post-filter settings. **(b)** Under EGPO, the all-zero group ratio decreases while the nonzero-advantage token ratio increases. **(c)** From a shared SFT initialization, EGPO reaches a peak validation avg@4 of 71.8 on AppWorld dev at step 60 (dashed line in **b**). Each method's best dev checkpoint is evaluated on test (Table 1).

Under binary terminal rewards, a group in which every rollout fails yields zero advantages and therefore provides no policy-gradient signal. Figure 4 reports the fraction of all-failure groups for Vanilla GRPO on AppWorld (N=8), averaged over training. For Qwen3-4B-Instruct, this fraction decreases from approximately 64% before SFT to 41% after SFT; for Qwen3-8B-Base, it decreases from approximately 67% to 55%. Consistently, Qwen3-4B-Instruct improves from 5.4% avg@4 zero-shot to 18.5% after SFT (Table 1).

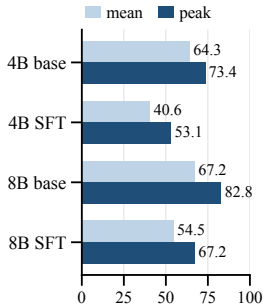


Figure 4: All-zero group ratio during GRPO.

EGPO consistently outperforms policy-optimization baselines across benchmarks and backbones. With Qwen3-4B-Instruct, AnyTool-Agent achieves 55.2% avg@4 and 77.4% best@4 on AppWorld, and 60.4% avg@4 and 74.0% best@4 on BFCL multi-turn (Table 1). Relative to Vanilla GRPO from the same SFT initialization, EGPO improves avg@4 by 5.5 and 5.7 percentage points on the two benchmarks, respectively. SFT alone achieves 18.5% avg@4 on AppWorld and 40.8% on BFCL, while the full pipeline improves AppWorld avg@4 by 49.8 points over the zero-shot model. The BFCL results use no human-annotated training trajectories. Under the same data and

Configuration	avg@4	best@4
SFT + Vanilla GRPO	49.7	70.8
+ Action Scoring	50.1	71.8
+ Scarcity Weighting	52.0	73.3
+ Multiplicative Reweighting	**55.2**	**77.4**

(a) Contribution of each EGPO component

Configuration	avg@4	best@4
Full EGPO	**55.2**	**77.4**
− API-hit term	54.8	72.6
− Format penalty	53.1	73.8
− Hallucination penalty	52.7	71.4

(b) Contribution of each action-score term

Table 2: Ablations on AppWorld with Qwen3-4B-Instruct. **(a)** Starting from SFT + Vanilla GRPO, sequentially add action scoring with a fixed coefficient, scarcity weighting of API hits, and multiplicative reweighting of A_o to obtain full EGPO (Eq. 4). **(b)** Remove each action-score term from full EGPO.

training protocol, EGPO also exceeds DAPO and GSPO, which achieve 51.8% and 52.5% avg@4 on AppWorld and 55.7% and 58.0% on BFCL, respectively. The gains persist with Qwen3-8B-Base: EGPO improves avg@4 over Vanilla GRPO from 36.6% to 47.8% on AppWorld and from 55.8% to 60.6% on BFCL, gains of 11.2 and 4.8 percentage points, respectively. These results should not be interpreted as a direct comparison between the two backbone sizes, because Qwen3-4B-Instruct is instruction-tuned whereas Qwen3-8B-Base is not.

EGPO continues improving on AppWorld. Starting from the same 18.5% SFT initialization, Vanilla GRPO's validation avg@4 on the AppWorld dev split plateaus after approximately 50 optimization steps, whereas EGPO keeps improving and peaks at step 60 (Figure 3c). Although the backbone and evaluation protocols are not directly comparable, AnyTool-Agent with Qwen3-4B-Instruct achieves 55.2% avg@4 and 77.4% best@4, numerically higher than the reported AgentEvolver results of 48.7% and 69.4%, respectively, obtained with a 14B backbone and eight rollouts (Table 11). We therefore report these numbers only as a reference. Additional comparisons with prior RL methods are provided in Appendix D.

3.3 Ablation Studies

We ablate EGPO's *action-level scoring*, *scarcity weighting*, and *advantage reweighting*. Table 2(a) evaluates their incremental gains and how action scores are combined with A_o. Table 2(b) removes individual action-score terms to assess their roles.

Action scoring and scarcity weighting provide complementary gains. Adding *Action Scoring* to Vanilla GRPO yields 50.1% avg@4 and 71.8% best@4, improvements of 0.4 and 1.0 percentage points, respectively. Thus, directly adding action-level scores to the outcome advantage provides limited gains. Adding *Scarcity Weighting* further improves performance to 52.0% / 73.3% (+1.9 / +1.5), indicating that emphasizing reference API hits visited by fewer rollouts provides additional discriminative signal. Nevertheless, this additive configuration remains 3.2 points in avg@4 and 4.1 points in best@4 below full EGPO.

Multiplicative reweighting improves credit assignment while preserving signs. Replacing the additive combination in the +Scarcity Weighting configuration with *Multiplicative Reweighting* yields full EGPO at 55.2% avg@4 and 77.4% best@4, the largest incremental improvement in Table 2(a) (+3.2 / +4.1). Both configurations use the same centered action scores, including the same execution rewards and penalties. The additive baseline applies these scores with a fixed coefficient, whereas EGPO couples their effect to A_o through multiplicative reweighting. Under additive reweighting, approximately 12% of action-level advantages have a sign opposite to the rollout outcome; among failed trajectories, this rate rises to 19% (Table 9 and Appendix E.3). EGPO yields no sign reversals. This characterizes the update rule, not the cause of the performance gap.

Action-score terms serve distinct roles. Removing the API-hit term reduces performance to 54.8% avg@4 and 72.6% best@4. The larger drop in best@4 (4.8 points) than in avg@4 (0.4 points) suggests that API matching particularly benefits the best of multiple attempts. Removing the format or hallucination penalty yields 53.1% / 73.8% and 52.7% / 71.4%, respectively. At step 50, these ablations raise the training outcome reward to 0.72 and 0.68, versus 0.65 for EGPO, yet reduce

validation best@4. This divergence suggests that the penalties curb malformed calls and invented API names that may boost training reward without improving task completion.

3.4 Analysis

EGPO yields larger gains on tasks that require modifying the environment. Relative to Vanilla GRPO from the same SFT initialization, EGPO improves avg@4 by 12.0, 7.9, and 7.2 percentage points on `file_system`, Venmo, and Spotify, respectively (Table 8). In these applications, successful completion requires API calls that produce observable changes in the sandbox. In contrast, EGPO decreases avg@4 by 9.4 points on `simple_note`, whose tasks involve few such calls. This pattern suggests that execution-grounded credit is most useful when sandbox feedback can distinguish actions that advance the task.

EGPO reallocates credit rather than expanding API coverage. Compared with Vanilla GRPO, EGPO reduces the redundant-loop rate from 10.2% to 6.4% (Table 9) and increases state-changing calls per trajectory from 2.44 to 3.25 (Table 10); the additive baseline yields an intermediate loop rate of 8.1%. Under EGPO, reference-API hit coverage reaches 0.66, and the least frequently visited reference APIs receive scarcity multipliers of up to $2.6\times$, while overall visited-API coverage changes only marginally. These results are consistent with EGPO shifting credit toward rare reference hits present in sampled rollouts, rather than substantially expanding the set of APIs visited.

EGPO is limited when execution feedback is uninformative or the outcome advantage vanishes. Repetitive information-retrieval loops account for 86.8% of the 38 test tasks unsolved by EGPO (Appendix C.5), suggesting that repeated calls with little state variation provide limited action-level feedback. Moreover, when $A_o = 0$, multiplicative reweighting leaves action-level advantages at zero. At the late checkpoint, the all-zero-group ratio is 0.16 under both the additive baseline and EGPO, compared with 0.41 under Vanilla GRPO (Table 9). Because the additive baseline attains the same ratio but lower performance, fewer all-zero groups alone cannot explain EGPO's gains. These findings highlight the complementary roles of the two stages: SFT increases the availability of nonzero outcome advantages, whereas EGPO redistributes them across actions.

4 Related Work

Reinforcement learning for tool use. Outcome-based RL advances tool use through methods such as LOOP and ReTool (Chen et al., 2025; Feng et al., 2025a). GRPO shares a group-relative trajectory advantage across generated tokens (Shao et al., 2024), without distinguishing individual action contributions. DAPO improves sampling and clipping (Yu et al., 2025), GSPO uses sequence-level importance ratios and clipping (Zheng et al., 2025), and RAGEN studies instability from reward-variance collapse (Wang et al., 2025). AgentEvolver and CoEvolve couple policy learning with task and data evolution (Zhai et al., 2025; Yang et al., 2026). Finer-grained approaches include ToolRL's reward-granularity analysis (Qian et al., 2025), process reward models for intermediate steps or tool calls (Wang et al., 2024b; Lightman et al., 2024; Ma et al., 2026), and AgentEvolver's combination of attribution and outcome rewards (Zhai et al., 2025). Others derive local credit from shared transitions (SALT (Li et al., 2026)), returns at recurrent anchor states (GiGPO (Feng et al., 2025b)), or entropy-guided branching (ARPO (Dong et al., 2026)). EGPO instead uses verified reference sequences and execution feedback to reweight outcome advantages across actions, preserving their signs and per-trajectory means without adding a separate process reward term.

5 Conclusion

We introduce AnyTool-Agent for adapting agents to new API ecosystems without manually annotated trajectories. API-Guided Data Synthesis provides SFT demonstrations and verified RL references, covering $2.4\times$ as many APIs as random exploration under the same generation budget. EGPO redistributes nonzero outcome advantages across actions using references and execution feedback, preserving signs and per-trajectory means. With Qwen3-4B, AnyTool-Agent achieves 55.2%/77.4% avg@4/best@4 on AppWorld and 60.4%/74.0% on BFCL multi-turn, improving avg@4 over Vanilla GRPO from the same SFT initialization by 5.5 and 5.7 percentage points, respectively.

AI USE STATEMENT

We used generative AI tools in two roles. First, as a writing aid: we used them to polish language, improve clarity, and check grammar in text the authors had already drafted. Second, as a coding assistant: we used them for code completion, refactoring, and debugging of the data-synthesis pipeline, the training scripts, and the plotting code. We did not use generative AI tools for research ideation, method design, experimental design, or the interpretation of results; all claims, analyses, and conclusions in this paper were formulated by the authors. All AI-assisted text was reviewed and edited by the authors, and all AI-assisted code was reviewed by the authors and validated by execution in the same sandbox environments used for our experiments. We take responsibility for the final content of this work, including its text, claims, and artifacts. Separately, a large language model (Qwen3-Max) serves as the expert model inside the proposed data-synthesis pipeline (§2.2); this is a component of the method under study, not an authoring aid.

REFERENCES

Emre Can Acikgoz, Cheng Qian, Jonas Hübotter, Heng Ji, Dilek Hakkani-Tür, and Gokhan Tür. Tool-R0: Self-evolving LLM agents for tool-learning from zero data. *arXiv preprint arXiv:2602.21320*, 2026.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers. In *International Conference on Learning Representations*, 2024.

Kevin Chen, Marco Cusumano-Towner, Brody Huval, Aleksei Petrenko, Jackson Hamburger, Vladlen Koltun, and Philipp Krähenbühl. Reinforcement learning for long-horizon interactive LLM agents. *arXiv preprint arXiv:2502.01600*, 2025.

Guanting Dong, Hangyu Mao, Kai Ma, Licheng Bao, Yifei Chen, Zhongyuan Wang, Zhongxia Chen, Jiazhen Du, Huiyang Wang, Fuzheng Zhang, Guorui Zhou, Yutao Zhu, Ji-Rong Wen, and Zhicheng Dou. Agentic reinforced policy optimization. In *International Conference on Learning Representations*, 2026.

Yu Du, Fangyun Wei, and Hongyang Zhang. AnyTool: Self-reflective, hierarchical agents for large-scale API calls. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235, pp. 11812–11829, 2024.

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*, 2025a.

Lang Feng, Zhenghai Xue, Tingcong Liu, and Bo An. Group-in-group policy optimization for LLM agent training. In *Advances in Neural Information Processing Systems*, 2025b. arXiv:2505.10978.

Bingguang Hao, Zengzhuang Xu, Yuntao Wen, Xinyi Xu, Yang Liu, Tong Zhao, Maolin Wang, Long Chen, Dong Wang, Yicheng Chen, Cunyin Peng, Xiangyu Zhao, Chenyi Zhuang, and Ji Zhang. From failure to mastery: Generating hard samples for tool-use agents. *arXiv preprint arXiv:2601.01498*, 2026.

Jiazheng Li, Yawei Wang, David Yan, Yijun Tian, Zhichao Xu, Huan Song, Panpan Xu, and Lin Lee Cheong. SALT: Step-level advantage assignment for long-horizon agents via trajectory graph. In *Findings of the Association for Computational Linguistics: EACL*, 2026. arXiv:2510.20022.

Xiaoxi Li, Wenxiang Jiao, Jiarui Jin, Guanting Dong, Jiajie Jin, Yinuo Wang, Hao Wang, Yutao Zhu, Ji-Rong Wen, Yuan Lu, and Zhicheng Dou. Deepagent: A general reasoning agent with scalable toolsets. *arXiv preprint arXiv:2510.21618*, 2025.

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

Boyin Liu, Zhuo Zhang, Sen Huang, Lipeng Xie, Qingxu Fu, Haoran Chen, Li Yu, Tianyi Hu, Zhaoyang Liu, Bolin Ding, and Dongbin Zhao. Voting with the graph: Stable RLAIF via topological consistency maximization. *arXiv preprint arXiv:2510.15514*, 2025.

Zuxin Liu, Thai Hoang, Jianguo Zhang, Ming Zhu, Tian Lan, Shirley Kokane, Juntao Tan, Weiran Yao, Zhiwei Liu, Yihao Feng, Rithesh Murthy, Liangwei Yang, Silvio Savarese, Juan Carlos Niebles, Huan Wang, Shelby Heinecke, and Caiming Xiong. APIGen: Automated pipeline for generating verifiable and diverse function-calling datasets. In *Advances in Neural Information Processing Systems*, volume 37, 2024.

Da Ma, Ziyue Yang, Hongshen Xu, Haotian Fang, Kai Yu, and Lu Chen. Empowering LLM tool invocation with tool-call reward model. In *International Conference on Learning Representations*, 2026.

Xinyi Ni, Haonan Jian, Qiuyang Wang, Vedanshi Chetan Shah, and Pengyu Hong. Doc2Agent: Scalable generation of tool-using agents from API documentation. *arXiv preprint arXiv:2506.19998*, 2025a.

Xinyi Ni, Qiuyang Wang, Yukun Zhang, and Pengyu Hong. ToolFactory: Automating tool generation by leveraging LLM to understand REST API documentations. *arXiv preprint arXiv:2501.16945*, 2025b.

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. Gorilla: Large language model connected with massive APIs. In *Advances in Neural Information Processing Systems*, volume 37, 2024.

Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E Gonzalez. The Berkeley function calling leaderboard (BFCL): From tool use to agentic evaluation of large language models. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267, pp. 48371–48392, 2025.

Akshara Prabhakar, Zuxin Liu, Ming Zhu, Jianguo Zhang, Tulika Awalgaonkar, Shiyu Wang, Zhiwei Liu, Haolin Chen, Thai Hoang, Juan Carlos Niebles, Shelby Heinecke, Weiran Yao, Huan Wang, Silvio Savarese, and Caiming Xiong. APIGen-MT: Agentic pipeline for multi-turn data generation via simulated agent-human interplay. *arXiv preprint arXiv:2504.03601*, 2025.

Zehan Qi, Xiao Liu, Iat Long Iong, Hanyu Lai, Xueqiao Sun, Wenyi Zhao, Yu Yang, Xinyue Yang, Jiadai Sun, Shuntian Yao, Tianjie Zhang, Wei Xu, Jie Tang, and Yuxiao Dong. WebRL: Training LLM web agents via self-evolving online curriculum reinforcement learning. In *International Conference on Learning Representations*, 2025.

Cheng Qian, Emre Can Acikgoz, Qi He, Hongru Wang, Xiusi Chen, Dilek Hakkani-Tür, Gokhan Tur, and Heng Ji. ToolRL: Reward is all tool learning needs. *arXiv preprint arXiv:2504.13958*, 2025.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *International Conference on Learning Representations*, 2024.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

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

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. HybridFlow: A flexible and efficient RLHF framework. In *Proceedings of the Twentieth European Conference on Computer Systems (EuroSys)*, 2025. arXiv:2409.19256.

Zexu Sun, Bokai Ji, Hengyi Cai, Shuaiqiang Wang, Lei Wang, Guangxia Li, and Xu Chen. AgentSkiller: Scaling generalist agent intelligence through semantically integrated cross-domain data synthesis. *arXiv preprint arXiv:2602.09372*, 2026.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. AppWorld: A controllable world of apps and people for benchmarking interactive coding agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, 2024.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linji Fan, and Anima Anandkumar. VOYAGER: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*, 2024a.

Peiyi Wang, Lei Li, Zhihong Shao, Runxin Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhifang Sui. Math-Shepherd: Verify and reinforce LLMs step-by-step without human annotations. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 9426–9439, 2024b.

Zihan Wang, Kangrui Wang, Qineng Wang, Pingyue Zhang, Linjie Li, Zhengyuan Yang, Xing Jin, Kefan Yu, Minh Nhat Nguyen, Licheng Liu, Eli Gottlieb, Yiping Lu, Kyunghyun Cho, Jiajun Wu, Li Fei-Fei, Lijuan Wang, Yejin Choi, and Manling Li. RAGEN: Understanding self-evolution in LLM agents via multi-turn reinforcement learning. *arXiv preprint arXiv:2504.20073*, 2025.

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

An Yang et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Shidong Yang, Ziyu Ma, Tongwen Huang, Yiming Hu, Yong Wang, and Xiangxiang Chu. CoEvolve: Training LLM agents via agent-data mutual evolution. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics*, 2026. arXiv:2604.15840.

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

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, et al. DAPO: An open-source LLM reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025.

Yunpeng Zhai, Shuchang Tao, Cheng Chen, Anni Zou, Ziqian Chen, Qingxu Fu, Shinji Mai, Li Yu, Jiaji Deng, Zouying Cao, Zhaoyang Liu, Bolin Ding, and Jingren Zhou. AgentEvolver: Towards efficient self-evolving agent system. *arXiv preprint arXiv:2511.10395*, 2025.

Chujie Zheng, Shixuan Liu, Mingze Li, Xiong-Hui Chen, Bowen Yu, Chang Gao, Kai Dang, Yuqiong Liu, Rui Men, An Yang, Jingren Zhou, and Junyang Lin. Group sequence policy optimization. *arXiv preprint arXiv:2507.18071*, 2025.

Appendix

A. Implementation Details 14
A.1 Hyperparameters 14
A.2 Action Score Details 14
A.3 Prompt Templates 15
A.4 Compute Cost and Reproducibility 18
B. Data and Benchmarks 19
B.1 Data Generation Pipeline 19
B.2 Reference Trajectory Construction 19
B.3 Dataset and Task Statistics 20
C. Behavioral Analysis 20
C.1 Per-Application Breakdown 20
C.2 Aggregate Behavioral Metrics 21
C.3 Case Study: Error Loops versus Successful Completion 22
C.4 Effects of Scarcity and Advantage Reweighting 23
C.5 Failure-Mode Taxonomy 24
D. Prior Reported RL Comparisons 25
E. Theoretical Analysis 25
E.1 All-Zero Groups Are Structural under Vanilla GRPO 25
E.2 EGPO Preserves Outcome-Determined Direction and Credit 26
E.3 Additive Process Rewards Can Conflict with Outcomes 26
E.4 Scarcity Weighting Reallocates Credit Across APIs 27
E.5 Why Targeted Selection Can Improve API Coverage 28
Related Work 28
Broader Impact 28

A Implementation Details

A.1 Hyperparameters

Table 3 lists the hyperparameters for the full AnyTool-Agent runs. Individual ablations change only the component identified in their labels.

Parameter	**Value**
RL Training	
Algorithm	GRPO
Normalization	Mean-only
Group size N	8
Train batch size	64
Learning rate	5×10^{-7}
KL penalty β	0.01
KL loss type	Forward KL
Validation frequency	Every 5 steps
Validation samples n	4
Gradient clipping	5.0
PPO clip ratio	0.2 / 0.3
EGPO-Specific	
Reweighting strength α	0.6
Scarcity cap κ	5.0
Additive score coefficient λ	0.6 (matched to α)
Action-score weights $w_{\text{api}}, w_{\text{hal}}, w_{\text{fmt}}$	1.0 each
Rollout	
Temperature	0.7
Max prompt length	4096
Max response length	20480
Max model length	32768
Tensor parallelism	2
GPU memory utilization	0.6
Max env workers	64
SFT Phase	
Epochs	5
Training samples	1,868
Infrastructure	
GPUs	8× NVIDIA A800
Framework	veRL (Sheng et al., 2025) + vLLM
FSDP optimizer offload	Enabled
Gradient checkpointing	Enabled

Table 3: Hyperparameters for the full AnyTool-Agent runs.

A.2 Action Score Details

We detail the deterministic execution signals used to estimate action-level credit. The additive baseline (**+Action Scoring** in Table 2a) clips these scores to $[-1, 1]$, centers them within the trajectory, and adds the result to the outcome advantage at $\lambda = 0.6$; EGPO consumes the same clipped and centered score and instead uses it to reweight the outcome advantage across actions. Signal weights are configured per run; we report the values used in the main experiments in Table 4.

API Hit Score. Given the reference action set $\tilde{\mathcal{A}}_{\text{GT}}$ (extracted from τ^*_{GT} and expanded with specification-query dependencies), each agent action at step t is matched against $\tilde{\mathcal{A}}_{\text{GT}}$. A first hit receives a positive score scaled by API type (full credit for write operations and discounted credit for read and specification-query APIs). A repeat hit receives $0.8\times$ its API-type score only when both its call arguments and resulting observation change, and zero otherwise (Table 4).

Signal	Score
First hit, write API	$+1.0$
First hit, read API	$+0.8$
First hit, specification-query API	$+0.5$
Repeat hit, changed arguments and substantively changed observation	$\times 0.8$
Repeat hit, otherwise	0
≥ 3 consecutive same-API calls or empty steps	-2
Error on a reference API	-1.0
Call to an API absent from the toolset	-1.0 (raises an error)
Format violation	-2
Terminal: successful submission ($R = 1$)	$+1.0$
Terminal: failed submission ($R = 0$)	-1.0
Terminal: failed submission within ≤ 5 steps ($R = 0$)	-1.5

Table 4: Deterministic action-score signals and the values used in the main experiments. Raw values are clipped to $[-1, 1]$ before centering.

Hallucination Penalty. The hallucination penalty covers execution-detected failures to advance the task: ≥ 3 consecutive calls to the same API or ≥ 3 empty steps incur a degenerate-loop penalty, and an error on a reference API incurs a fixed penalty. Calls to APIs absent from the toolset are not scored separately; they raise an error and receive the same penalty.

Format Penalty. Actions that do not contain exactly one valid API-call block incur a format penalty.

Terminal Judgment. The `complete_task` action triggers the deterministic AppWorld evaluator, which returns a binary outcome $R \in \{0, 1\}$. A successful submission assigns $+1.0$ to the terminal step. A failed submission assigns a base score of -1.0, increased in magnitude to -1.5 when it occurs within five steps. The `complete_task` action carries no base API-hit credit.

AppWorld Grader. At RL scoring time, AppWorld uses deterministic environment feedback to evaluate executed calls; task completion is determined by the environment evaluator when the agent invokes `complete_task`.

A.3 Prompt Templates

We provide paper-formatted versions of the prompts used in AppWorld experiments, preserving their inputs, decision rules, scoring ranges, and output schemas while omitting implementation-specific serialization details. BFCL uses the standard multi-turn function-calling prompt from (Patil et al., 2025) without modification.

System prompt. The AppWorld system prompt establishes the agent as an autonomous assistant operating a Python REPL environment on the user's behalf, and bootstraps the agent with three discovery APIs (`show_app_descriptions`, `show_api_descriptions`, `show_api_doc`) that the agent must call to learn the available tools; there is no upfront tool list. The full template:

```
I am your supervisor and you are a super intelligent AI
Assistant whose job is to achieve my day-to-day tasks
completely autonomously. To do this, you will need to
interact with app/s (e.g., spotify, venmo etc) using
their associated APIs on my behalf. For this you will
undertake a *multi-step conversation* using a python
REPL environment. That is, you will write the python
code and the environment will execute it and show you
the result, based on which, you will write python code
for the next step and so on, until you've achieved the
goal.

# Three discovery APIs:
```

```
print(apis.api_docs.show_app_descriptions())
print(apis.api_docs.show_api_descriptions(app_name='X'))
print(apis.api_docs.show_api_doc(app_name='X', api_name='Y'))

Each code execution produces an output for subsequent
calls. My name / email / phone are provided per task.
Task: <task instruction>
```

Per-step reasoning directive. To elicit code-aligned reasoning at each step, we append the following directive to the system prompt during both SFT data generation and RL rollout (so the SFT and RL distributions match). The directive asks for a first-person intention immediately before each code block, with the API named in the intention required to match the one in the code block, which avoids the inconsistency between free-form reasoning and executed code that arises when reasoning is generated independently:

```
## Reasoning requirement (per-step intention)
Before EACH Python code block, write a first-person
intention inside <thinking>...</thinking>. State what
this step does and why, and where each key argument
comes from (user-provided / a value quoted from a
specific previous API response / computed -- show the
logic).
Rules:
- The API you mention in thinking MUST match the API
  in the code block. Do NOT explore alternative paths
  in thinking -- commit to the action you will take.
- After </thinking>, give the ```python code block.
```

We deliberately impose no length constraint on the thinking block; the model expands or contracts it according to the reasoning demand of each step, which avoids both the over-long boilerplate of fixed-length templates and the under-reasoning of a hard cap.

Semantic verifier prompt. During data synthesis, Qwen3-Max assigns each expert trajectory a semantic verification score $s_{\text{judge}} \in [0, 1]$. The verifier receives the synthesized task, a reference solution, and the complete execution trajectory. It returns an integer score in $[0, 100]$, which is divided by 100. Candidate tasks pass semantic post-filtering when $s_{\text{judge}} \geq 0.85$. This score is used only during data synthesis and is distinct from the binary environment outcome $R \in \{0, 1\}$ used during RL and evaluation. We use Qwen3-Max with temperature 0.7 and a maximum output length of 8,192 tokens. The prompt below preserves the complete evaluation procedure and scoring contract while omitting implementation-specific serialization details.

```
You are an expert evaluator of tool-using agents.
Evaluate whether the agent successfully completes the
user task.

# Inputs

## User Task
{task}

## Reference Solution
{reference_solution}

## Agent Trajectory
The trajectory contains the agent's actions and the
resulting environment observations in chronological
order.

{trajectory}
```

```
# Evaluation Rules

Base the evaluation strictly on the provided trajectory.
Do not assume or invent unobserved actions or results.
Use the reference solution to check correctness and
execution efficiency, but accept alternative solutions
that achieve the same valid outcome. When identifying an
error, cite the corresponding action or observation.

Follow these steps:

1. Relevance
Determine whether the trajectory is materially related
to the user task. If it is entirely unrelated, assign a
score of 0 and stop.

2. Repetition
Check whether the agent repeats the same or nearly
identical action at least three times without obtaining
new information or making progress. If this occurs in an
otherwise relevant trajectory, the final score must not
exceed 20.

3. Goal achievement
Examine the complete trajectory and final result.
Determine whether all requirements in the user task were
satisfied correctly. Compare both the execution path and
the final result with the reference solution, while
allowing other valid solutions.

In particular, verify:
- whether the required APIs were called;
- whether API arguments were correct and complete;
- whether paginated results were handled when necessary;
- whether observations support the claimed result;
- whether the final state satisfies the task; and
- whether failed, skipped, or invalid steps prevent
  completion.

Do not treat confident language or an unsupported final
answer as evidence of successful execution.

4. Execution quality
Deduct points for runtime errors, failed tool calls,
invalid code, unnecessary actions, redundant exploration,
or substantial inefficiency relative to the reference
solution. Calls made only to inspect API documentation
are not considered unnecessary.

# Scoring

If the goal is achieved, assign a score from 60 to 100:
- 90--100: Correct, complete, and efficient, with no
  significant issue.
- 80--89: Correct, with only minor errors or
  inefficiencies.
- 70--79: Correct, but noticeably inefficient or with
  several unnecessary steps.
```

Run	Wall-clock (h)	GPU-hours
SFT (1,868 traj., 5 epochs)	$\sim$2.6	21
Vanilla GRPO (80 steps, full run)	$\sim$52	416
EGPO (to peak step 60)	$\sim$39	312
Total (full pipeline)	$\sim$42	**333**

Table 5: Compute cost for the main AppWorld experiments (8$\times$A800). GPU-hours = wall-clock $\times$ 8. Total counts the SFT stage plus the EGPO RL path (Vanilla GRPO is a control, not an additive stage). EGPO and Vanilla GRPO cost about the same per step ($\sim$0.65 h/step): the deterministic action-score computation adds negligible overhead ($<$0.5 s/step) relative to rollout generation. The two RL rows therefore differ in how many steps they report, not in cost per step; wall-clock figures are indicative and subject to run-to-run scheduling variation.

```
- 60--69: Correct, but with substantial execution or
  clarity issues.

If the goal is not achieved, assign a score from 0 to 40:
- 30--40: Incorrect, but showing meaningful progress
  along a valid path.
- 10--29: Incorrect, with major execution problems and
  limited progress.
- 1--9: Incorrect, with only minimal task-relevant
  progress.
- 0: Entirely irrelevant or unsuccessful.

Scores from 41 to 59 are not permitted. If repeated
actions trigger the repetition rule, the score must not
exceed 20.

# Output Format

First provide a concise analysis grounded in specific
actions and observations. Then output exactly one integer
score using:

<reward>SCORE</reward>
```

A.4 Compute Cost and Reproducibility

Hardware and framework. All experiments run on a single node of 8$\times$NVIDIA A800 GPUs (80 GB each) using veRL (Sheng et al., 2025) with vLLM for rollout, FSDP for the actor, and gradient checkpointing plus optimizer offload to fit the 4B/8B backbones within memory. Tensor parallelism is 2 for the rollout engine.

Training wall-clock. Table 5 reports the wall-clock cost of the main AppWorld runs. SFT on 1,868 trajectories (5 epochs) takes $\sim$2.6 hours. The EGPO run reaches its reported peak at step 60 in $\sim$39 hours (group size $N = 8$, batch 64); the Vanilla GRPO control runs the full 80 steps in $\sim$52 hours, i.e. the same cost per step. Each training step takes roughly 2,300–5,000 s, dominated by rollout generation and the periodic validation pass; the actor update itself is only a few hundred seconds. We report a model FLOP utilization (MFU) of $\sim$0.33 and a peak GPU memory of 77.9 GB, leaving headroom on the 80 GB devices.

Data generation cost. API-Guided Data Synthesis produces 4,932 candidate tasks and 8,018 sandbox trajectories (Table 6) using a Qwen3-Max expert model via DashScope; this stage is API-bound, costing roughly 3 GPU-hours of the local sandbox plus $\sim$90,000 Qwen3-Max API calls ($\sim$670M DashScope tokens), with no human annotation.

Stage	Count
Candidate tasks generated	4,932
Tasks after post-filtering	3,696
Trajectories collected	8,018
with nonzero LLM-judge score	4,181
SFT trajectories (after extraction)	1,868

Table 6: Data generation pipeline statistics on AppWorld.

Reproducibility. The main results report the mean over three random seeds. Training uses a group size of 8, a learning rate of 5×10^{-7}, and KL $\beta = 0.01$ (full hyperparameters in Appendix A.1). Validation uses $n = 4$ rollouts at temperature 0.7 every 5 steps. We will release the synthesized data, the SFT/RL checkpoints, and the training scripts upon publication.

Anonymous code availability. An anonymized implementation is available at `https://anonymous.4open.science/r/anytool-agent-D029/`.

B Data and Benchmarks

B.1 Data Generation Pipeline

Task Synthesis. Because API specs enumerate the available capabilities, the selector can directly specify which API combinations each task should cover. It samples targets from three strategies: (1) single-app tasks involving 1–3 APIs within one application, (2) cross-app tasks requiring coordination across 2–4 applications, and (3) complex tasks involving 4+ applications with data dependencies. Selecting targets this way directs the synthesis distribution toward desired API regions instead of inheriting the visitation bias of a random walk. Tasks then refer to records that exist in the database by relationship, time, or status rather than by literal value, requiring the agent to resolve references through API interaction.

Filtering Criteria. A candidate is retained only if it satisfies the same two checks as in §2.2: the sandbox run completes with consistent state transitions, and a separate model confirms semantic completion of the instruction. The pruned core sequence is then re-executed in a fresh sandbox; only a sequence that re-executes successfully is kept as τ^*_{GT}.

B.2 Reference Trajectory Construction

Extraction. τ^*_{GT} is derived from a successful expert (Qwen3-Max) trajectory that passes two verification rounds: a verification rollout of the expert trajectory, followed by re-execution of the extracted core steps in a fresh sandbox. The verified core API-call sequence is serialized as Python code (`task.ground_truth`), from which we extract four sets via regular expression: full API names `gt_apis` (e.g., `apis.venmo.login`), app names, API names, and exact (app, api) pairs.

Expansion to $\tilde{\mathcal{A}}_{\text{GT}}$. The expanded reference action set adds the three specification-query APIs by rule rather than call-graph analysis: `show_app_descriptions` (always), `show_api_descriptions` (if any reference app exists), and `show_api_doc` (if any reference (app, api) pair exists). A step counts as a reference hit if its `apis.X.Y` is in `gt_apis`, or its specification-query arguments match a reference app or pair.

Contents. τ^*_{GT} stores the ordered API-call sequence and the exact (app, api) pairings (preventing cross-app contamination); per-API type weights (state-changing = full credit, information-retrieval = discounted, specification-query = lowest). Expected state differences are not stored in τ^*_{GT} but are enforced by AppWorld's `evaluation.py` at grading time.

Split	Scenarios	Tasks	Used for
Train (official)	35	90	In RL pool
Synthesized	–	60	In RL pool
Dev	19	57	Checkpoint selection
Test-normal	56	168	Final evaluation

Table 7: AppWorld splits used in this work. The RL training pool is the 90 official train tasks plus 60 synthesized tasks constructed from that split and the public API specs (150 total). The dev split (57 tasks) is used only for checkpoint selection, and the test-normal split (168 tasks $\times$ 4 rollouts) is used for final evaluation.

B.3 Dataset and Task Statistics

AppWorld. AppWorld (Trivedi et al., 2024) provides 9 interconnected applications (Amazon, Gmail, Phone, Spotify, Venmo, etc.) exposing 457 APIs over a shared SQLite-backed state. Amazon is among the nine but is not the primary application of any test task in Table 8, which therefore reports eight apps. Table 7 summarizes the splits. We use the official dev split (57 tasks) exclusively for checkpoint selection and report final performance on the test-normal split (168 tasks). To rule out data leakage, all synthesized tasks, training trajectories, and τ^*_{GT} are derived exclusively from the train split and the public API specs, and no dev or test task is used for training. The RL training pool is the 90 official train tasks plus 60 synthesized tasks constructed from that split (150 tasks total).

BFCL multi-turn. We evaluate on the four multi-turn subsets of BFCL (Patil et al., 2025): `base`, `missing-function`, `missing-parameter`, and `long-context`. The four subsets contain 200 instances each (800 official evaluation instances in total), with matched indices representing variants of the same 200 underlying task scenarios. These average 2–5 tool calls per task and are graded by a state-based multi-turn checker, providing a shorter-horizon transfer setting where SFT already lowers the rate of all-zero groups and EGPO mainly improves the allocation of the available outcome advantage. The official evaluation queries and reference solutions are excluded from both SFT and RL training. All policy-optimization methods share the same SFT initialization and synthetic RL pool. We select checkpoints by avg@4 on a separate synthetic validation set before evaluating them on the official BFCL instances.

Synthesized data. Table 6 reports the pipeline yields: 4,932 candidate tasks, of which 3,696 (75%) survive sandbox execution and semantic verification; 8,018 sandbox trajectories, of which 4,181 receive a nonzero LLM-judge score and 1,868 serve as SFT data. A separate subset of 60 synthesized tasks joins the 90 official train tasks in the RL pool, each with a reference trajectory τ^*_{GT}. The synthesized share of the RL pool is deliberately kept at this size. Synthesized tasks are constructed inside the existing train-split sandboxes rather than in sandboxes purpose-built for them, so their state configurations are less varied than those of the official train tasks and each supplies less outcome contrast per task. In our pipeline the synthesized data acts through the SFT initialization that makes outcome-based RL viable (§3.2) and through the reference trajectories EGPO scores against, not through the size of the RL pool.

C Behavioral Analysis

C.1 Per-Application Breakdown

Table 8 disaggregates the AppWorld test-set TGC by application. EGPO improves avg@4 on 5 of 8 apps. The largest gains are on file_system (+12.0 pp), Venmo (+7.9), and Spotify (+7.2). Gmail also gains on avg@4 (+8.3) but loses on best@4 (46.7 to 33.3); with 15 tasks a single task moves best@4 by 6.7 pp, so we do not count it among the gains. Two apps regress: simple_note (-9.4 pp), whose tasks involve few state-changing operations so the action score provides limited additional signal, and splitwise (-2.1), where both methods remain near 6–8% and the difference lies within evaluation noise. Phone is unchanged. Multi-app tasks (31 tasks spanning $\geq$2 apps) gain only +1.6 pp avg@4.

App	n	EGPO avg@4	EGPO best@4	Vanilla avg@4	Vanilla best@4	Δavg
file_system	27	34.3	48.1	22.2	44.4	**+12.0**
venmo	63	51.6	74.6	43.7	66.7	**+7.9**
spotify	45	69.4	91.1	62.2	82.2	**+7.2**
phone	22	55.7	95.5	55.7	86.4	0.0
gmail	15	23.3	33.3	15.0	46.7	**+8.3**
todoist	9	2.8	11.1	0.0	0.0	+2.8
splitwise	12	6.2	25.0	8.3	25.0	−2.1
simple_note	24	53.1	70.8	62.5	83.3	−9.4
All (Test)	168	**55.2**	**77.4**	**49.7**	**70.8**	**+5.5**

Table 8: AppWorld test-set TGC disaggregated by application (Qwen3-4B-Instruct). A multi-app task is counted in each involved app. Amazon is not the primary application of any test task, so the table has eight rows. EGPO improves avg@4 on 5 of 8 apps; the largest gains are on file_system, Venmo, and Spotify. simple_note and splitwise regress; phone is unchanged (see text). The aggregate row matches Table 1.

Metric	Vanilla	+Action Scoring	EGPO
All-zero group ratio↓	0.41	**0.16**	**0.16**
τ^*_{GT} hit coverage↑	N/A	0.44	**0.66**
Adv. sign-flip ratio↓	N/A	0.12	**0.00**
Hallucinated-call rate↓	0.000	0.002	**0.000**
Redundant-loop rate↓	10.2%	8.1%	**6.4%**

Table 9: Summary behavioral metrics at the late checkpoint of each run, taken at the final training step; all runs start from the same SFT initialization. **+Action Scoring** = the same clipped, within-trajectory-centered action score as EGPO, added to A_o at $\lambda = 0.6 = \alpha$ instead of multiplying it. The all-zero group ratio and the τ^*_{GT} hit coverage are per-batch values at that step; the hit coverage is the fraction of reference APIs that the rollout calls, averaged over trajectories with a non-empty reference. The sign-flip ratio is the fraction of action advantages whose sign disagrees with the trajectory outcome, measured over the full run; on failed trajectories the **+Action Scoring** rate rises to 0.19 (Appendix E.3). The hallucinated-call rate is measured per call and rounded to three decimals, so 0.000 under EGPO is consistent with the three task-level hallucination failures in §C.5. N/A marks a metric undefined for Vanilla GRPO, which has neither an action score nor reweighting: it records no API-hit signal, and with no action-level modulation every action advantage trivially shares the sign of A_o.

C.2 Aggregate Behavioral Metrics

Table 9 summarizes late-checkpoint behavioral metrics; Table 10 reports mid-training rollout statistics. Together with the case studies in §C.3, they show where EGPO shifts the policy. Two mid-training rows in Table 10 move in the opposite direction from the late-checkpoint redundant-loop rate and merit clarification. The repetition trigger fires on ≥ 3 consecutive calls to the same API regardless of whether the observation changes (§A.2), so it also counts legitimate sequences such as reading several API documentations in a row; EGPO issues more specification-query calls per trajectory (5.72 vs. 4.32) and more API calls overall (8.67 vs. 7.66), which raises both rows. The redundant-loop rate in Table 9 is measured on the late checkpoint and counts degenerate loops that do not advance the task; that rate falls from 10.2% to 6.4%. The all-zero-advantage group ratio at the late checkpoint is 0.41 under Vanilla GRPO and 0.16 under both **+Action Scoring** and EGPO (Table 9). The drop coincides with the introduction of an action-level signal and not with the accuracy that signal delivers: **+Action Scoring** reaches the same ratio while gaining 0.4 avg@4 points over Vanilla GRPO, against 5.5 for EGPO. We report this as an observed pattern and do not attribute it to a specific mechanism. EGPO itself does not act on all-zero groups: when the mean-centered outcome advantage is zero, multiplicative reweighting preserves that zero signal (Theorem 2).

Metric (per trajectory)	Vanilla	EGPO
API calls	7.66	8.67
Trajectory length (steps)	14.7	16.5
Specification-query calls	4.32	5.72
State-changing (write) calls	2.44	3.25
Visited API coverage	0.798	0.816
Same-API consecutive (mean)	2.54	3.05
Repetition-trigger ratio	0.338	0.422
Hallucination rate	0.000	0.000
Outcome reward (mean)	0.257	0.275

Table 10: Behavioral metrics on AppWorld training rollouts (Vanilla at step 24, EGPO at step 21; both from the SFT-initialized checkpoint). A single step may issue more than one call (§2.3), so each row is a per-trajectory tally of its own category and the rows are not a partition of the trajectory. Sign-flip under EGPO is 0.00 throughout (Table 9).

C.3 Case Study: Error Loops versus Successful Completion

For qualitative analysis, we examine two instantiated variants of the same AppWorld request-approval task template and report four rollouts per method. Each variant requires the agent to identify specified contact groups through Phone, retrieve pending payment requests from Venmo, and approve only the matching requests. We first compare EGPO with the additive baseline and then compare Vanilla GRPO with EGPO (Figure 5).

+Action Scoring: wrong-API loops, all four fail. Across its four rollouts, **+Action Scoring** selects an incorrect operation in 2 of 4 rollouts. These trajectories invoke the valid but goal-reversing `create_payment_request`, which creates rather than approves a request, or attempt nonexistent variants such as `send_payment_request` and `send_money`, instead of the required `approve_payment_request`. Each rollout attempts `complete_task` once but fails the unit tests; trajectories are markedly longer (mean 29,360 tokens vs. 22,063 for EGPO, both averaged over the four rollouts of that method), reflecting cycles through incorrect operations before submission. Locally valid steps, such as logging in and retrieving contacts or pending requests, contribute positive additive score terms even though the trajectory fails. Because these terms are not constrained by the outcome, they weaken, and can in general overturn, the negative update supplied by the final result.

EGPO: succeeds. Under multiplicative reweighting, every step advantage remains tied to the trajectory-level outcome advantage (Eq. 4). A successful EGPO rollout follows the required cross-application dependencies, approves all three matching requests, and reaches `complete_task` with all unit tests passing in 11 API calls. Because a step in a trajectory headed for failure receives an attenuated negative (not positive) gradient, there is no incentive to commit to operations that do not advance the task.

Advantage sign consistency across training. The additive baseline and EGPO differ in the sign consistency of their updates. Under the additive baseline, 11.9% of action advantages carry a sign opposite to their trajectory outcome, and 18.6% do so on failed trajectories (Appendix E.3); the locally valid steps of the failing rollouts above are of the kind that receive such updates. Under EGPO the ratio is 0.00 throughout training (Table 9), as Theorem 2 requires. We report this sign-consistency contrast as an observed difference between the runs without claiming that it causes the performance gap.

Vanilla GRPO vs. EGPO: redundant-query loops vs. concise success. A second failure mode appears under Vanilla GRPO (no action-level signal), which has the highest redundant-loop rate among all methods (10.2%, Table 9). On the second request-approval instance, all four Vanilla GRPO rollouts fail by cycling among valid but task-irrelevant transaction queries (`show_transactions` and `show_transaction`) and hallucinated retrieval variants (`get_transactions`, `get_transaction_history`, and `list_payments`). One rollout makes 16 such read-oriented API attempts without invoking the required `approve_payment_request`, while producing longer thinking blocks (mean 12,984 tokens across all Vanilla GRPO roll-

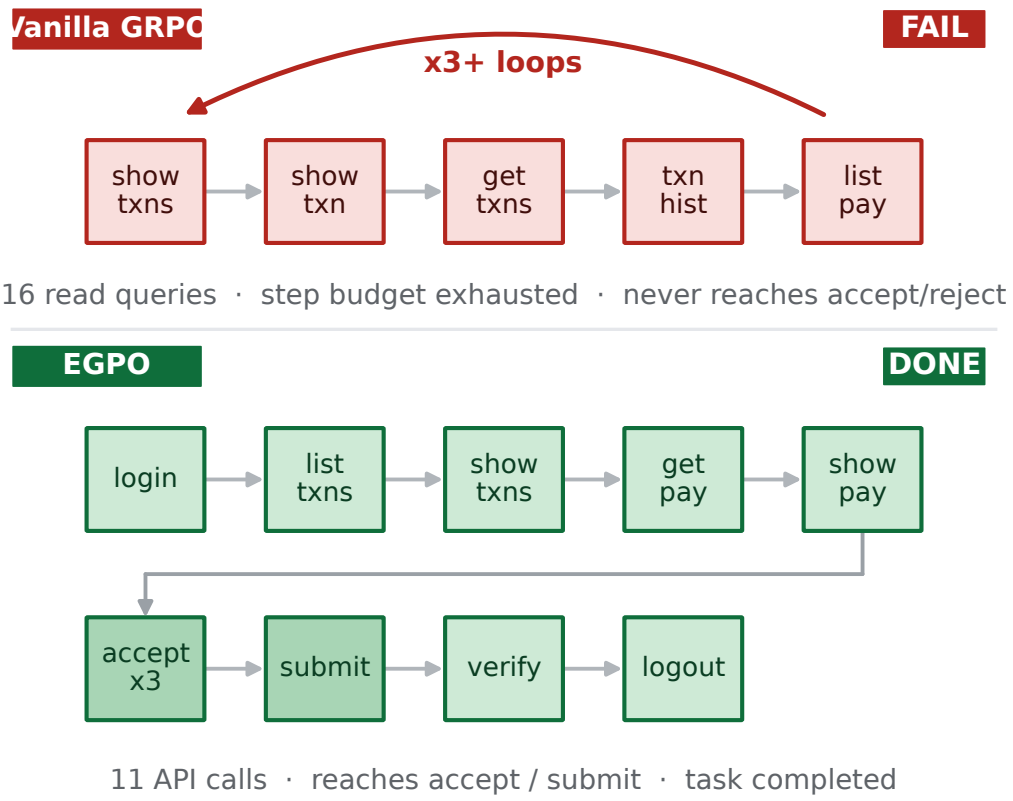
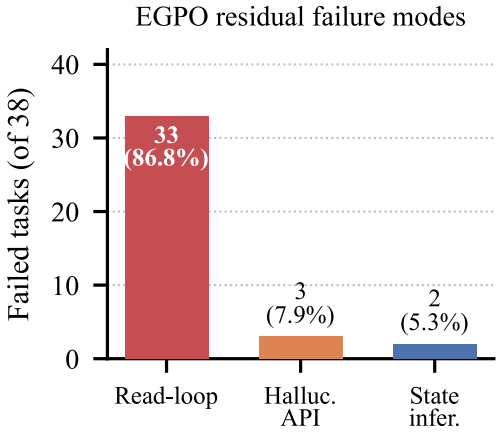


Figure 5: A representative AppWorld request-approval instance. Vanilla GRPO cycles among task-irrelevant transaction queries and hallucinated retrieval APIs for 16 attempts without invoking `approve_payment_request`, then exhausts the step budget. EGPO follows the task dependencies (`show_profile` → `show_account_passwords` → `phone.login` → `phone.search_contacts`×2 → `venmo.login` → `venmo.show_received_payment_requests` → `venmo.approve_payment_request`×3 → `complete_task`) and completes in 11 API calls. Vanilla fails by cycling rather than by insufficient deliberation.

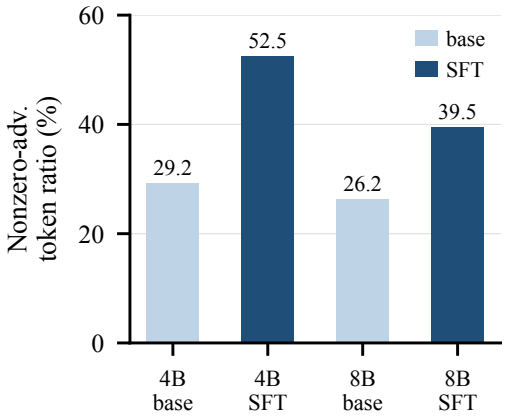
outs vs. 7,919 for the successful EGPO rollout). The EGPO rollout, by contrast, identifies the relevant contact groups with `phone.search_contacts`, retrieves pending requests with `venmo.show_received_payment_requests`, approves the three matching requests with `venmo.approve_payment_request`, and completes the task in 11 API calls. Vanilla fails by cycling through redundant and invalid query attempts, which EGPO's hallucination penalty discourages ($r_{\text{hal}}^{(t)}$ in §2.3). This per-task pattern is the qualitative signature of the aggregate redundant-loop-rate contrast in Table 9 (Vanilla GRPO 10.2% → **+Action Scoring** 8.1% → EGPO 6.4%). The hallucination term is the component of EGPO's action score that targets such loops; we report the rate contrast as an observed difference between the runs and do not isolate that term's individual contribution.

C.4 Effects of Scarcity and Advantage Reweighting

Scarcity weighting reshapes gradient allocation, not call coverage. The **+Scarcity Weighting** configuration improves avg@4 by 1.9 pp over **+Action Scoring** (Table 2a). Scarcity weighting does not materially expand raw API-call coverage (Appendix C.2). Instead, it changes how credit is allocated among APIs already reached during sampling: the largest observed multiplier is about 2.6× for the rarest APIs, 87.7% of steps carry a non-zero multiplier, and the mean multiplier is 1.02 (Appendix E.4). The multiplier distribution is the direct measurement of the scarcity term, and it supports an effect on credit allocation rather than the creation of new signal on APIs that no rollout reaches.



(a) EGPO residual failure modes on the 38 AppWorld test tasks it still fails (best@4 < 1.0). Redundant query loops account for 86.8%, indicating the residual weakness is concentrated in a single structural mode rather than scattered.



(b) Fraction of tokens receiving nonzero policy-advantage signals during Vanilla GRPO training on AppWorld for Qwen3-4B-Instruct and Qwen3-8B-Base, with and without SFT; fewer than 30% of tokens receive any nonzero advantage without SFT.

Figure 6: Residual failure modes of EGPO and signal availability under Vanilla GRPO.

Advantage reweighting aligns action-level signals with the outcome. Additive use of the action score can favor locally valid but task-irrelevant operations independently of the final result: removing multiplicative reweighting (**+Action Scoring**) places more action-level score on incorrect or penalized operations and leaves a higher redundant-loop rate (8.1%) than full EGPO (6.4%), while Vanilla GRPO, with neither action-level evidence nor action-level allocation, loops most (10.2%). The observed advantage sign-flip ratio is 0.00 under EGPO, where no locally correct step in a globally failing trajectory receives a positive gradient, against 0.12 under **+Action Scoring** (Table 9). Qualitatively, **+Action Scoring** trajectories invoke goal-reversing or nonexistent operations (e.g., creating rather than approving a payment request) and cycle through incorrect operations before a failed submission; under EGPO the same prompt yields task completion (Appendix C.3). As the leave-one-out ablation shows (Table 2b), format and hallucination signals also regularize against degenerate shortcuts.

C.5 Failure-Mode Taxonomy

We categorize the 38 AppWorld test tasks EGPO still fails (best@4 < 1.0):

- **(a) Repetitive query loops** (33 tasks, 86.8%): all four rollouts exhaust the step budget while repeatedly issuing information-retrieval calls (e.g., `show`, `list`, `get`, `search`). For instance, `9ef798c_1` (Venmo) repeatedly calls `show_received_payment_requests` despite an insufficient-balance error until the step budget is exhausted.
- **(b) Hallucinated APIs** (3, 7.9%): calls to non-existent methods, raising `AttributeError`. For instance, `b6d1104_1` invokes a non-existent simple_note method.
- **(c) Multi-step state inference** (2, 5.3%): the task flow completes but the final answer or environment state is incorrect. For instance, `166f4ff_1` sums Venmo requests to \$133 versus a ground-truth of \$471 (omitted requests or a misread time range).

The concentration on a single identifiable mode (a) indicates EGPO's residual weakness is structural rather than scattered: EGPO lowers the redundant-loop rate (10.2%→6.4%, Table 9) yet does not fully resolve loops when an environment error leaves the policy trapped. Richer execution feedback for such errors is the clearest future direction.

Method	Model	AppWorld TGC (%)		
		TGC@1	avg@4	best@4
AgentEvolver (Zhai et al., 2025)	Qwen2.5-7B	–	32.4‡	51.2‡
AgentEvolver (Zhai et al., 2025)	Qwen2.5-14B	–	48.7‡	69.4‡
CoEvolve (Yang et al., 2026)	Qwen3-4B	35.7	–	–
SALT (Li et al., 2026)	Qwen2.5-32B	66.2	–	–
LOOP (Chen et al., 2025)	Qwen2.5-32B	71.3	–	–
AnyTool-Agent	Qwen3-4B-Instruct	–	**55.2**	**77.4**

Table 11: Prior RL methods reported on AppWorld (not directly comparable: heterogeneous protocols/backbones, auxiliary modules beyond the policy; typically without released code or weights). AnyTool-Agent uses our protocol (avg/best@4, n=4, temp 0.7). TGC@1 = single-rollout TGC (temp 0). ‡over 8 rollouts (original protocol).

D Prior Reported RL Comparisons

Table 11 contextualizes AnyTool-Agent against other RL methods reported on AppWorld. These results use heterogeneous backbones, protocols, and auxiliary modules, so the numbers provide background rather than a controlled comparison.

These numbers are contextual only. AgentEvolver couples the policy with auxiliary data-evolution modules; CoEvolve couples RL with self-evolving data synthesis (same Qwen3-4B backbone, but single-rollout TGC protocol); SALT is a plug-and-play step-level advantage module on top of GRPO with a 32B backbone; LOOP uses a 32B backbone with extra components. Our controlled study isolates EGPO on Qwen3-4B-Instruct under a fixed protocol.

E Theoretical Analysis

We provide formal justifications for the central claims of the paper. Throughout, we consider a group of N rollouts $\{\tau^{(b)}\}_{b=1}^{N}$ on the same prompt, with outcome rewards $R^{(b)} \in [0,1]$, group mean $\bar{R}$, and outcome advantage $A_o^{(b)} = R^{(b)} - \bar{R}$ as in §2. Theorems 1 and 3 additionally assume binary outcomes, $R^{(b)} \in \{0,1\}$, as given by the AppWorld TGC metric; the remaining results hold for $R^{(b)} \in [0,1]$.

E.1 All-Zero Groups Are Structural under Vanilla GRPO

We show that all-zero groups are not an artifact of a particular model but a structural property of Vanilla GRPO on long-horizon tasks.

Theorem 1 (Probability of an all-zero advantage group)**.** *Let each rollout succeed independently with probability $p \in (0,1)$, and let the outcome reward be binary ($R^{(b)} \in \{0,1\}$). The probability that all N rollouts in a group share the same outcome (and hence receive zero mean-centered advantage) is*

$$P(\textit{all-zero}) = p^N + (1-p)^N.$$

In a stylized long-horizon model in which the T steps of a task succeed independently with constant probability ρ and the task succeeds only if all steps succeed, $p = \rho^T$, giving

$$P(\textit{all-zero}) = \rho^{TN} + (1-\rho^T)^N \xrightarrow{T\to\infty} 1,$$

since $\rho^T \to 0$ and $(1-\rho^T)^N \to 1$ for fixed N.

Proof. Under independent rollouts, the number of successes $K \sim \text{Binomial}(N,p)$. A group has non-zero advantage iff $0 < K < N$ (at least one success and one failure). Thus $P(\text{non-zero}) = 1 - P(K{=}0) - P(K{=}N) = 1 - (1-p)^N - p^N$, and $P(\text{all-zero}) = (1-p)^N + p^N$. Substituting $p = \rho^T$ yields the claim; as $T \to \infty$, $p \to 0$, so $p^N \to 0$ and $(1-p)^N \to 1$. □

Implication. For Qwen3-4B-Instruct, substituting its zero-shot success rate ($p = 0.054$, Table 1) and our group size $N = 8$ gives $P(\text{all-zero}) = 0.64$, matching its empirical pre-SFT all-zero ratio of

approximately 64% in Figure 4. This calculation illustrates why all-zero groups become common when per-rollout success is low. The corresponding nonzero-advantage token ratio is shown in Figure 6b.

E.2 EGPO Preserves Outcome-Determined Direction and Credit

A natural concern is whether action-level feedback can reverse the update direction set by A_o. We show that EGPO preserves this direction when its multiplicative weights remain positive.

Theorem 2 (Sign preservation and credit conservation)**.** *Let $r^{(t)}_{soft} \in [-1, 1]$ (clipped), let $\bar{r}_{soft} = \frac{1}{T}\sum_{t=1}^{T} r^{(t)}_{soft}$ be the within-trajectory mean, and define the data-dependent radius $\Delta := \max_{1\le t\le T} |r^{(t)}_{soft} - \bar{r}_{soft}| \le 2$. With reweighting strength α satisfying $\alpha\Delta < 1$, the multiplicative weight satisfies $1 + \delta^{(t)} \in [1 - \alpha\Delta,\, 1 + \alpha\Delta] \subset (0, \infty)$, hence*

$$\operatorname{sgn}\big(A^{(t)}_s\big) = \operatorname{sgn}(A_o) \cdot \operatorname{sgn}(1 + \delta^{(t)}) = \operatorname{sgn}(A_o) \quad \forall\, t,$$

since $1 + \delta^{(t)} > 0$. Consequently, no trajectory with $A_o < 0$ ever receives a positive step advantage. Mean centering further gives $\sum_{t=1}^{T} A^{(t)}_s = TA_o$ for a trajectory of T actions.

Proof. $\delta^{(t)} = \alpha \cdot \operatorname{sgn}(A_o) \cdot (r^{(t)}_{\text{soft}} - \bar{r}_{\text{soft}})$, so $|\delta^{(t)}| \le \alpha\Delta$. With $\alpha\Delta < 1$, $1 + \delta^{(t)} \ge 1 - \alpha\Delta > 0$, hence $\operatorname{sgn}(1 + \delta^{(t)}) = +1$ and $\operatorname{sgn}(A^{(t)}_s) = \operatorname{sgn}(A_o)$. For a trajectory of T actions, mean centering gives $\sum_{t=1}^{T}(r^{(t)}_{\text{soft}} - \bar{r}_{\text{soft}}) = 0$ and therefore $\sum_t \delta^{(t)} = 0$. It follows from Eq. 4 that $\sum_{t=1}^{T} A^{(t)}_s = A_o \sum_{t=1}^{T}(1 + \delta^{(t)}) = TA_o$. This identity concerns the action-level advantages defined by EGPO; it does not assert conservation of token-weighted gradient norms. □

Implication. The outcome advantage determines both the direction of the update and the sum of step advantages; action-level feedback changes only the distribution among actions.

Remark. The bound $\alpha\Delta < 1$ is data-dependent: Δ is the empirical radius of the centered action score, measurable from training rollouts. In our experiments $\alpha = 0.6$, and the centered action score is highly concentrated near zero (intra-trajectory mean centering removes the bulk of the offset), so $\alpha\Delta < 1$ holds in practice. We verify this directly: across all training rollouts of the main AppWorld run, the 99th-percentile radius is $\Delta_{99} = 1.49$ ($\alpha\Delta_{99} = 0.894 < 1$), and the mean is $\Delta_{\text{mean}} = 1.01$; the worst-case $\Delta_{\text{max}} = 1.92$ ($\alpha\Delta_{\text{max}} = 1.15$) occurs in only 0.004% of trajectories. On that 0.004% the sufficient condition $\alpha\Delta < 1$ fails and the theorem gives no guarantee; a flip would there additionally require $\operatorname{sgn}(A_o)\,(r^{(t)}_{\text{soft}} - \bar{r}_{\text{soft}}) < -1/\alpha$ at the extremal step, which the sign of the deviation need not satisfy. The distribution is stable across training steps (Δ_{mean} varies from 1.02 to 1.01 from step 1 to step 80), and no flip is observed: the advantage sign-flip ratio is 0.00 to two decimals at every training step (Table 9). The theorem thus gives the regime where sign preservation is guaranteed, and our configuration lies inside it for all but a negligible fraction of trajectories.

E.3 Fixed-Scale Additive Process Rewards Can Conflict with Outcomes

We analyze the fixed-scale additive alternative $A^{(t)}_s = A_o + \lambda\,(r^{(t)}_{\text{soft}} - \bar{r}_{\text{soft}})$. Because it applies the centered action score with an outcome-independent coefficient, a sufficiently large local score can reverse the update direction determined by A_o. EGPO instead couples action-level modulation to A_o: execution rewards and penalties rescale the outcome advantage rather than acting as an independent offset. The **+Action Scoring** ablation implements the fixed-scale additive rule with $\lambda = 0.6$ and uses the same clipped, centered action scores as full EGPO.

Theorem 3 (Fixed-scale additive combination admits sign violation)**.** *Consider the additive combination $A^{(t)}_s = A_o + \lambda\,(r^{(t)}_{soft} - \bar{r}_{soft})$ with $\lambda > 0$ and binary outcomes, where $r^{(t)}_{soft}$ is clipped to $[-1, 1]$ and centered within the trajectory exactly as in Eq. 3. In a group of N rollouts of which exactly one succeeds, every failed trajectory has $A_o = -1/N$, so $A^{(t)}_s > 0$ at any step whose centered score exceeds $1/(\lambda N)$. Sign agreement is therefore guaranteed only when λ is small enough that $\lambda\Delta < \min_b |A^{(b)}_o|$, where $\Delta := \max_t |r^{(t)}_{soft} - \bar{r}_{soft}| \le 2$. Since a single-success group gives*

$\min_b |A_o^{(b)}| = 1/N$, *any* $\lambda < 1/(2N)$ *preserves sign for every score configuration; at* $N = 8$ *this requires* $\lambda < 0.0625$*, an order of magnitude below the* $\lambda = 0.6$ *used in our experiments. A uniform sign-preservation guarantee therefore requires a substantially smaller coefficient: with* $\lambda = 0.6$ *the threshold* $1/(\lambda N) = 0.208$ *lies well inside the attainable range* $[-2, 2]$*, and neither clipping nor within-trajectory centering removes the resulting violation.*

Proof. With binary outcomes and exactly one success among N rollouts, $\bar{R} = 1/N$, so a failed trajectory τ has $A_o(\tau) = 0 - 1/N = -1/N$. Let t be a step of τ whose centered score satisfies $r_{\text{soft}}^{(t)} - \bar{r}_{\text{soft}} > 1/(\lambda N)$. Then $A_s^{(t)}(\tau) = -1/N + \lambda\,(r_{\text{soft}}^{(t)} - \bar{r}_{\text{soft}}) > 0$, while the trajectory failed. Such a step is admissible: within-trajectory centering removes only the trajectory-level offset, so it suppresses the case in which an entire failing rollout scores uniformly high, but not the case in which one step scores far above the mean of its own rollout—a correct login, or a correct first hit on a reference API, inside an otherwise failing trajectory. Since the centered score ranges over $[-2, 2]$, the threshold $1/(\lambda N)$ is attainable whenever $\lambda N > 1/2$. The distinction between the evaluated rules is structural: the fixed-scale additive baseline applies the centered score as an outcome-independent offset, whereas EGPO couples the same signal to A_o through the factor $1 + \delta^{(t)}$. When this factor stays positive under Theorem 2, it fixes $\text{sgn}(A_s^{(t)}) = \text{sgn}(A_o)$ at every step. □

Implication. The theorem does not claim that every fixed-scale additive update flips sign. Instead, it shows that guaranteeing agreement with the final outcome for all admissible score configurations requires $\lambda < 1/(2N)$, whereas EGPO preserves sign whenever the condition of Theorem 2 holds. The violation occurs at a measurable rate: throughout the **+Action Scoring** run, 11.9% of all action advantages disagree in sign with their trajectory outcome; restricted to failed trajectories the rate is 18.6%, and restricted to steps of failed trajectories that carry a nonzero action score it is 29.5%. Under full EGPO the same measurement returns 0.00 at every checkpoint (Table 9). These measurements confirm that the fixed-scale additive configuration operates in the sign-violating regime identified by the theorem, whereas full EGPO maintains outcome-aligned action advantages throughout training.

E.4 Scarcity Weighting Reallocates Credit Across APIs

A concern is whether scarcity weighting (Eq. 2) can reverse the credit implied by other signals. We show that it is a non-negative reweighting that changes the relative emphasis placed on APIs without changing the sign of an API-hit contribution.

Theorem 4 (Non-negative scarcity reweighting)**.** *Let* $\mu_c \in [0, 1]$ *be the within-group mean of the per-rollout hit indicators for API c, and* $\omega_c = \text{clip}((1 - \mu_c)/(\sigma_c + \epsilon), 0, \kappa)$*. Then* $\omega_c \in [0, \kappa]$*, so multiplying an API-hit signal by* ω_c *cannot reverse its sign. For a mastered API (*$\mu_c = 1$*),* $\omega_c = 0$*; for a rarely hit API with a sufficiently large unclipped ratio,* $\omega_c = \kappa$*. Thus scarcity weighting can change the relative magnitude, but not the direction, of API-hit contributions.*

Proof. The clipping interval directly gives $0 \le \omega_c \le \kappa$. Hence $\text{sgn}(\omega_c r_{\text{api}}^{(t)})$ is either $\text{sgn}(r_{\text{api}}^{(t)})$ or zero. Substituting $\mu_c = 1$ makes the numerator zero, while any unclipped value above κ is mapped to the cap. Scarcity weighting therefore rescales the API-hit component without reversing its sign. The subsequent advantage reweighting preserves the sign of the final step advantage when its weights are positive (Theorem 2). □

Implication. This explains the empirical result in §3.4: scarcity weighting does not change which APIs the policy eventually calls (raw call coverage is unchanged) but concentrates gradient on rare APIs already reached by at least one rollout, which surfaces as task-success gains rather than coverage changes. Scarcity weighting produces signal only for APIs reached by at least one rollout; if all rollouts miss API c, then $r_{\text{api}}^{(t)} = 0$ for all t, so its scarcity-weighted contribution is zero. We verify this distribution empirically: across all training rollouts, 87.7% of steps carry a non-zero multiplier ($\omega_c > 0$, corresponding to $\mu_c \in (0, 1)$), while 12.3% carry no signal (all-miss or all-hit APIs); the mean multiplier is 1.02 and the largest observed multiplier is 2.65 for the rarest APIs, well below the cap $\kappa = 5.0$, confirming that scarcity concentrates gradient on hard-to-learn APIs without affecting the majority of steps.

E.5 Why Targeted Selection Can Improve API Coverage

We use a stylized sampling model to explain why selecting target APIs from specifications can improve coverage over random exploration. This asymptotic analysis describes the capability of a coverage-directed selector; it does not predict the finite-budget $2.4\times$ measurement in Figure 3.

Theorem 5 (Coverage complexity: random vs. targeted)**.** *Let the tool environment contain K distinct APIs. (i) Under random exploration, where each generated task's solution hits one uniformly random API, covering all K APIs requires $\Theta(K \log K)$ tasks in expectation (coupon collector's bound). (ii) An idealized selector that schedules at least one not-yet-covered target API per accepted task covers all K APIs in $O(K)$ tasks. The asymptotic ratio between these idealized regimes is therefore $\Omega(\log K)$.*

Sketch. (i) Random exploration is the coupon collector problem: each task collects one uniformly random coupon (API), and $E[\text{tasks to cover } K] = KH_K = \Theta(K \log K)$, with the long tail (last few APIs) dominating. (ii) Under the stated scheduling assumption, each accepted task adds at least one new target API, so at most K accepted tasks are required; this gives the upper bound $O(K)$. Dividing the expectation in (i) by this upper bound gives a ratio of at least $\Omega(\log K)$. □

Implication. The analysis isolates the benefit of targeted selection: once API specs expose individual capabilities, a selector can direct synthesis toward a desired coverage distribution rather than wait for random paths to reach it. Our implementation uses targeted intra- and cross-app combinations rather than the idealized one-new-API schedule, and Figure 3 provides the corresponding finite-budget evidence: 325 APIs versus 135 for random exploration at 800 generated candidates per method.

Related Work

Tool-use data synthesis. Constructing executable training tasks for APIs unseen during training follows two routes. The top-down route generates queries first: APIGen (Liu et al., 2024), APIGen-MT (Prabhakar et al., 2025) and Toucan (Xu et al., 2025) generate user queries from API or MCP specifications, then obtain trajectories by execution or teacher models (Sun et al., 2026; Hao et al., 2026). On new APIs, many queries are not executable, and few tasks remain after verification. The exploration route recovers tasks from trajectories: AgentEvolver's self-questioning (Zhai et al., 2025) and Voyager (Wang et al., 2024a) first explore the environment and then synthesize tasks from the resulting trajectories (Acikgoz et al., 2026; Qi et al., 2025). Those trajectories are executable, but exploration repeatedly visits APIs that are already easy to reach and cannot specify which capabilities to cover. A related line exposes tool specifications to generation (Cai et al., 2024; Ni et al., 2025b;a), yet still does not determine task coverage. API-Guided Data Synthesis selects target combinations from the specifications, constructs tasks bottom-up, and retains a solution only when it passes a semantic completion check and re-executes in a fresh sandbox. The retained solutions become queries and reference trajectories τ^*_{GT}. Coverage is determined by the selected combinations, not by the paths that exploration happens to visit.

Broader Impact

AnyTool-Agent trains agents that execute state-changing APIs (payments, emails, file writes). Our evaluation uses fully simulated sandboxes (AppWorld/BFCL) with fictitious personas and no real user data, so the direct societal risk is limited; but deploying such agents on live tool ecosystems requires strict action-level authorization and rollback, which our framework does not address. Positively, EGPO's deterministic execution-feedback scores replace per-step LLM-as-judge calls, reducing cost ($<$0.5 s/step versus an additional LLM call per step) and removing a source of judge noise and non-reproducibility. This is a step toward auditable agent RL.