PARAAGENT: REINFORCING PARALLEL ACTING IN OPEN-WORLD TOOL ENVIRONMENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Language model agents are increasingly deployed in open-world tool environments, which require balancing exploring unknown capabilities and exploiting known ones. Existing methods face a performance–efficiency tradeoff: they either rigidly decouple exploration and execution or interleave them without coordination. We argue that the key lies not in whether to decouple or interleave them, but in how to coordinate them across granularities. We introduce PARAACT, a structured parallel-action loop that combines phase-level Exploration $\rightleftharpoons$ Execution with action-level parallelism. To learn this loop, PARAAGENT combines multi-agent cold-start demonstrations with reinforcement learning under multi-level advantage decoupling, making planning structure explicit and supervising it with step-, phase-, and trajectory-level rewards. Learning is supported by our TOOLENV, a scalable simulator grounded in 50,011 realistic tool interfaces. On two open-world tool benchmarks, PARAAGENT$_{4B}$ achieves the best average success among all baselines, including GPT-4.1 systems, with the largest gains on multi-tool tasks. Behavioral analyses show that these gains stem from this action organization, highlighting its importance for capable and efficient open-world agents.

1 INTRODUCTION

Recent progress in large language models (LLMs) has accelerated the transition from chatbots toward general agentic assistants (Wei et al., 2026; Guo et al., 2025; Zheng et al., 2025). Alongside this, the external tool ecosystem is rapidly expanding in scale and diversity, spanning APIs (Qin et al., 2024), MCP (Hou et al., 2025), and Skills (Zhang et al., 2025a). This is propelling LLM agents into **open-world tool environments**, where task constraints and tool utility often emerge progressively through interaction rather than specified in advance (Yao et al., 2023). Effective operation therefore requires balancing *exploration* of unfamiliar capabilities with *exploitation* of acquired information.

Existing approaches broadly follow two paradigms, as shown in Figure 1. *(1) Exploration-then-Execution* (Li et al., 2023; Qu et al., 2024; Kachuee et al., 2025; Huang et al., 2026) first explores a potential tool space, and then acts within this fixed context. Such decoupling enables tailored phase-wise optimization, facilitating more effective action strategies. However, early exploration errors or emergent requirements may remain unaddressed and ultimately limit task success. *(2) Exploration-and-Execution* can flexibly perform two types of actions, but existing methods (Qin et al., 2025; Li et al., 2026) naively interleave them as homogeneous actions in a flat sequence. Without explicit phase structure or coordination, this flexibility relies on frequent serial interleaving, accumulating irrelevant context over time. This exposes a clear **performance-efficiency tradeoff**: the former is efficient but brittle, whereas the latter is adaptive but interaction heavy.

The key is not whether to separate or interleave exploration and execution, but how to coordinate them across granularities, *phase-level adaptation* with *action-level parallelism*. Two challenges arise in **(i) Structure**: exploration is *epistemic*, reducing uncertainty about required capabilities, whereas execution is *operational*, coordinating under dependency. This asymmetry requires specialized parallel strategies and phase transitions when capability gaps emerge. **(ii) Learning**: the loop relies on structural decisions that in-context or imitation learning can initialize, but robustly optimizing them requires interaction (Zhang et al., 2025b). It further requires rewarding how actions are organized, as such decisions are unobservable in individual actions, with effects emerging only phases later.

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.

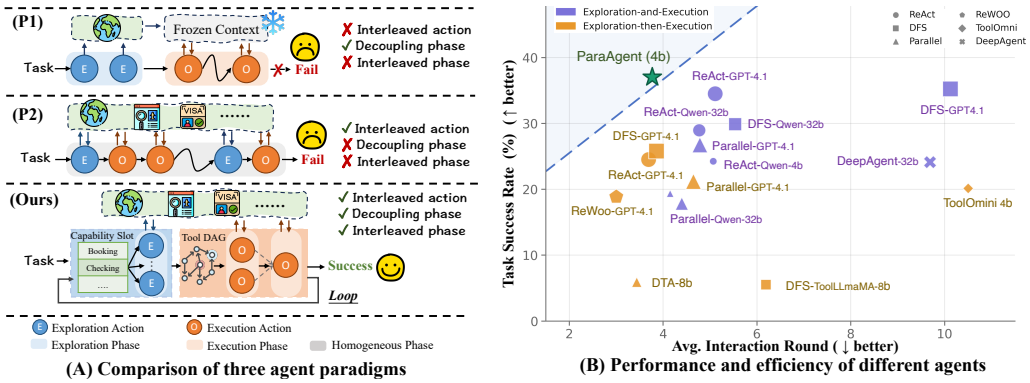


Figure 1: **Agent paradigms and their performance–efficiency tradeoff.** (A) *Exploration-then-Execution* (P1) rigidly decouples exploration and execution, while native *Exploration-and-Execution* (P2) interleaves them without coordination. Ours separate phase-level adaptation from action-level coordination. (B) Task success (Pass@1) vs. interaction rounds on ToolBench and API-Bank.

To address **(i)**, we introduce **PARAACT**, a **structured parallel-action loop** framework that integrates verbal reasoning, explicit planning, and parallel actions. At the phase level, it drives an *Exploration* $\rightleftharpoons$ *Execution* phase loop to expand capabilities as requirements evolve. At the action level, two primitives match the asymmetry: *Exploration parallelism* infers required capability slots and issues parallel queries to fill them. *Execution parallelism* selects acquired tools and organizes their dependencies as a directed acyclic graph (DAG) for parallel execution. Together, this framework supports phase-level adaptation while coordinating parallel actions within each phase.

To address **(ii)**, we further introduce **PARAAGENT**, an **agentic learning** framework that learns the structured parallel-action loop in open-world tool environments. PARAAGENT first bootstraps the loop through a cold-start stage using a state-sharing multi-agent system: a *planner* controls phase transitions based on a *critic*'s evidence-gap assessment, while a *retriever* and *executor* implement the two parallel primitives. Although these demonstrations initialize the loop, they cannot ensure robust structural decisions under evolving interactions, while standard GRPO (Shao et al., 2024) relies on outcome rewards that fail to capture action organization and dependencies. Hence, we introduce a reinforcement learning stage with multi-level advantage decoupling. Our key idea is to make implicit planning structure explicit through phase-level plans, making structural decisions observable and supervisable at the appropriate granularity. It comprises two key components: **(1) Hierarchical rewards.** *Step rewards* assess compliance with the plan protocol and formatting requirements; *phase rewards* evaluate planning and execution against the subgraph induced by declared tools in our global tool-dependency graph; and *trajectory rewards* assess final task outcomes via LLM- or rule-based evaluation together with reference-tool hit rate. Because phase rewards rely on self-generated plans, positive phase rewards are further gated by the final task outcome. **(2) Decoupled optimization.** Since naively aggregating heterogeneous rewards may obscure individual learning signals, we independently normalize three reward components, thereby balancing supervision across levels. The resulting learning signal links action organization to task outcomes, guiding the policy to refine its structural decisions through interaction.

To support this interactive training, we build TOOLENV, a scalable tool environment grounded in real-world tool interactions. We curate a large-scale corpus of *request–response pairs* and, following prior tool-world-model approaches (Guo et al., 2024; Li et al., 2025c), fine-tune LLMs on these pairs and combine them with caching and rule-based methods into a realistic, scalable testbed.

Extensive experiments on ToolBench and API-Bank compare PARAAGENT against baselines spanning diverse action strategies, including *serial*, *tree*, and *parallel* acting. PARAAGENT$_{4B}$ achieves the best average success, outperforming the strongest baseline and even GPT-4.1-based systems, with the largest gains on multi-tool tasks. Behavior analyses further show that these gains come from better action decisions rather than more interaction, and ablations validate the effectiveness of our two-stage learning and decoupled hierarchical rewards. Experimental results demonstrate the effectiveness and efficiency of learning our structured parallel-action loop across action strategies and model scales.

2 Related Work

Open-World Tool Use. Real-world agents increasingly operate in large, heterogeneous, and evolving tool environments, where required capabilities are unknown a priori and must be discovered during interaction (Qin et al., 2024; Hou et al., 2025; Xu et al., 2025; Zhou et al., 2026; Li et al., 2023). Existing methods broadly follow two paradigms. *Exploration-then-Execution* (Zhang et al., 2023; Patil et al., 2024; Xu et al., 2024; Kachuee et al., 2025; Qin et al., 2024) retrieves a fixed tool set before execution, with improvements through query rewriting, tool-document augmentation (Zhang et al., 2023; Xu et al., 2024; Kachuee et al., 2025; Shi et al., 2025), or dedicated tool retrievers (Qin et al., 2024). While enabling stage-specific optimization, fixing exploration before execution limits adaptation to emerging requirements. *Exploration-and-Execution* (Qin et al., 2025; Li et al., 2026; Huang et al., 2026) instead dynamically discovers and invokes tools during reasoning. Meta-Tool (Qin et al., 2025) integrates retrieval into tool-use reasoning, ToolOmni (Huang et al., 2026) jointly optimizes proactive retrieval and grounded execution, and DeepAgent (Li et al., 2026) unifies tool discovery and execution within an end-to-end reasoning process. However, these methods conflate phase-level and action-level coordination, limiting both efficiency and flexibility. ParaAgent separates phase-level adaptation from action-level coordination.

Agentic Learning. Agentic learning (Wei et al., 2026; Zeng et al., 2024; Qu et al., 2025; Yue et al., 2025; Zhang et al., 2025b) enhances LLMs to autonomously reason and act through environment interaction. Prompt-based methods (Yao et al., 2023; Shinn et al., 2023; Xu et al., 2023) enable tool use by interleaving reasoning and actions, but remain bounded by in-context capabilities. Demonstration fine-tuning (Qin et al., 2024; Zeng et al., 2024; Yue et al., 2025; Chen et al., 2024) improves task-specific behavior, yet depends on demonstration coverage and quality. Reinforcement learning (Shao et al., 2024; Guo et al., 2025; Wang et al., 2025; Wei et al., 2025) instead optimizes task success directly through interaction. However, existing RL methods face two limitations in open-world settings: *environment-wise*, current environments focus on closed, sequential tool interactions (Jin et al., 2025; Feng et al., 2025; Qian et al., 2026), lacking realistic simulation of large heterogeneous tool spaces; *algorithm-wise*, flat outcome rewards provide little supervision for phase transitions or parallel action structure (Huang et al., 2026; Li et al., 2026). ParaAgent addresses both: ToolEnv provides realistic open-world API simulation, while hierarchical multi-level rewards supervise action format, phase transitions, and task outcomes.

3 Methodology

3.1 Problem Statement

We consider an agent operating in an open-world tool environment $\mathcal{L}$. Given task τ, the agent acts over $\mathcal{A} = \mathcal{A}^e \cup \mathcal{A}^o$: *exploration* phase $\mathcal{A}^e$ issues parallel retrieval queries to discover relevant tools from $\mathcal{L}$, and *execution* phase $\mathcal{A}^o$ invokes the discovered tools concurrently under an inferred dependency. The two phases form a structured *Exploration* $\rightleftharpoons$ *Execution* loop, producing a trajectory:

$$\xi = (\mathcal{A}^e_1, o^e_1, \mathcal{A}^o_1, o^o_1, \ldots, \mathcal{A}^e_t, o^e_t, \mathcal{A}^o_t, o^o_t, \hat{y}), \tag{1}$$

where o^e_t and o^o_t are the retrieval and execution observations at loop t, and $\hat{y}$ is the final answer. The objective is to learn π_θ that maximizes task success $\mathbb{E}_{\xi\sim\pi_\theta}[\mathbf{1}[\hat{y} \text{ solves } \tau]]$.

3.2 ParaAct: A Structured Parallel Action Loop Framework

ParaAct structures the interaction process by three layers of verbal reasoning (<think>), structured plans (<plan>), and parallel actions (<search_tool> or <tool_call>). Exploration aims to identify capability gaps and determine which tools to search for, whereas execution aims to infer dependencies among acquired tools and schedule them efficiently. This distinction yields different planning structures: exploration enumerates capability slots to be searched in parallel; execution schedules concurrent calls based on inferred dependencies. If tools are unusable or capability needs remain unmet, the agent starts a new loop, allowing the working tool set to expand incrementally. After multiple loops, the answer is given within the <answer> and </answer> tags.

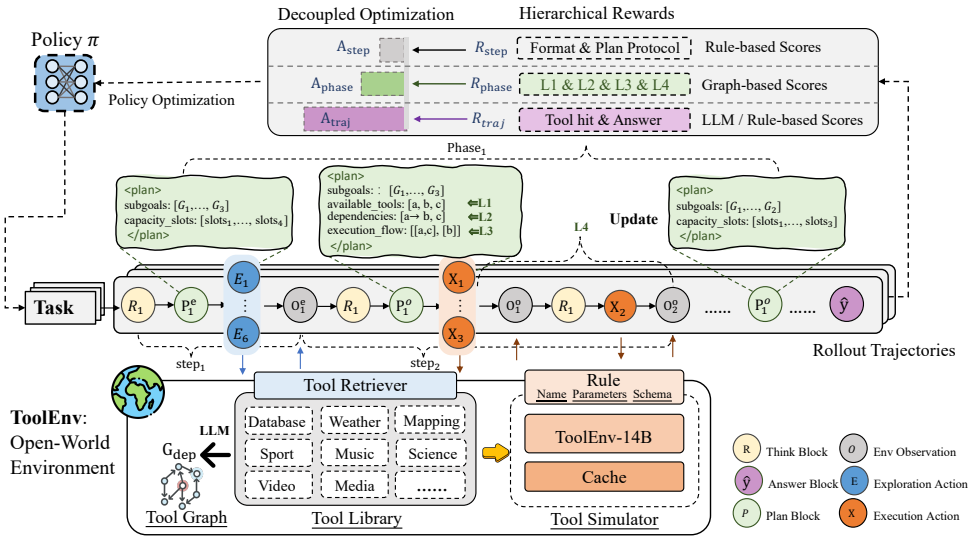


Figure 2: **Overview of PARAAGENT.** *Middle*: The agent generates rollouts in TOOLENV through a structured parallel-action loop. *Top*: A hierarchical reward provides supervision at step, phase, and trajectory levels, optimizing policy π via decoupled advantage. *Bottom*: TOOLENV is an open-world environment comprising a tool library, retriever, and three-layer simulator.

Exploration Parallelism. $\mathcal{A}^e$ first reasons in <think> and </think> tags to identify functional requirements the task needs, then organizes them into structured *capability slots* within the <plan>. Next, the plan simultaneously drives parallel search actions $\{q_1, \ldots, q_j\}$, and each contains a description of the required functionality in a <search_tool> and </search_tool> tag. Each q_i retrieves the top-k candidates from $\mathcal{L}$, yielding up to $j \times k$ new tools at loop t and updating the accumulated working tool set as:

$$\mathcal{B}_t = \mathcal{B}_{t-1} \cup \bigcup_{i=1}^{j} \mathrm{TopK}(\mathcal{L}, q_i), \qquad \mathcal{B}_0 = \varnothing. \tag{2}$$

This enables targeted search over capability gaps rather than undirected exploration. Based on the current slot coverage and accumulated observations, the policy decides whether to continue exploration or transition to execution.

Execution Parallelism. Some tools in $\mathcal{B}_t$ may be irrelevant or unusable, *e.g.*, due to a missing API key. Therefore, $\mathcal{A}^o$ first reasons within <think> to identify usable tools and infer their dependencies, and then organizes them into a structured <plan> with three components: (1) *Available tools*: the relevant subset of $\mathcal{B}_t$. (2) *Dependencies*: inter-call relationships modeled as a DAG $G_t = (\mathcal{V}_t, \mathcal{D}_t)$, where each node represents a uniquely identified call to an available tool, and $(l_i, l_j) \in \mathcal{D}_t$ indicates that l_j requires an output or state change produced by l_i. (3) *Execution flow*: a layer-wise topological schedule derived from G_t. Each layer $\mathcal{V}_s^{\|}$ consists of all tools with zero in-degree in the residual graph and is dispatched as a single concurrent batch via <tool_call>:

$$\mathcal{V}_s^{\|} = \{l \in \mathcal{V}_t : \deg^-(l) = 0\}, \quad G_t \leftarrow G_t \setminus \mathcal{V}_s^{\|}, \tag{3}$$

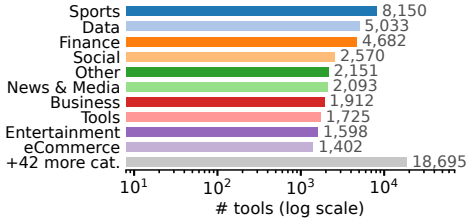
where $\mathcal{V}_t$ denotes the call nodes remaining in G_t and is updated as nodes are removed. Each result o^o feeds into dependent calls as input, and execution continues until all planned calls are completed. Structuring the plan explicitly elicits this reasoning from the model, while its phase-typed schema makes exploration and execution distinguishable, enabling targeted optimization for each phase.

3.3 PARAAGENT: LEARNING THE PARALLEL ACTION LOOP

To learn this structured parallel acting loop, we adopt a two-stage learning. Cold-start bootstrapping first instils the behavior from high-quality demonstrations. Online reinforcement learning, guided by multi-level advantage decoupling, then refines the policy through environment interaction.

$\mathcal{G}_{\text{dep}}$ statistic	Value
# Tools (nodes)	50,011
# Tools w/ dependency	33,020
# Directed edges	147,503
– Strong (HARD)	43,087
– Soft (SOFT)	104,416
Mean out-degree (connected)	4.47
Max out-degree	232
Max in-degree	131

(a) Dependency graph $\mathcal{G}_{\text{dep}}$



(b) Tool-library categories

Figure 3: (a) Statistics of $\mathcal{G}_{\text{dep}}$, with required HARD dependencies and non-mandatory SOFT orderings. (b) Top 10 of 52 categories by tool count in a long-tailed library.

Data Collection. **(1) Training data.** We collect source training data from four datasets covering three tool types and diverse task structures: API-based tasks (ToolBench (Qin et al., 2024), Simia (Li et al., 2025d)), MCP-based tasks (Toucan (Xu et al., 2025)), and deep research tasks (AFM (Li et al., 2025b)). We stratify tasks by difficulty, measured by the number of required tools, and bias the RL set toward harder, tool-intensive tasks. **(2) Tool dependency graph.** To support dependency-aware execution, we construct a global *tool dependency graph* $\mathcal{G}_{\text{dep}}$ over the complete tool library (§3.4). For each tool pair (l_i, l_j) and each direction of their interaction, we adopt a voting scheme of four distinct LLM[1] judges to label the relationship between l_j and l_i as *HARD* (strongly dependent), *SOFT* (conditionally related), or *NONE* (independent). Graph statistics are summarized in Figure 3(a). See Appendix A for full data statistics and graph construction details.

Cold-Start Bootstrapping. We construct a *state-sharing multi-agent system* to generate high-quality trajectories, providing a behavioral prior. The system consists of five specialized agents coordinated by a shared state pool: (1) *Planner* decomposes tasks, generates slots or DAGs, and routes modes. (2) *Retriever* performs slot-guided parallel retrieval. (3) *Executor* traverses the DAG in topological order using Kahn's algorithm (Kahn, 1962) and dispatches independent tool calls in parallel. (4) *Critic* provides transition supervision only during trajectory generation; at inference time, the learned policy makes phase-transition decisions directly. (5) *Answerer* synthesizes verified observations into the final answer. All agents produce both reasoning and structured actions. The *Planner* routes the system across four modes: Exploration, Execution, Refine, and Answer. After each phase, the *Critic* evaluates coverage to determine the next transition. After format validation, the generated trajectories are serialized and used to supervised fine-tune the base model. See Appendix B for full details of the multi-agent system design.

Hierarchical Reward. Since our parallel action loop structures the trajectory into phases and steps with distinct behavioral objectives, a single terminal reward is insufficient: it cannot attribute failures to specific malformed action steps, cannot guide the model to learn when to transition between exploration and execution phases, and provides no signal for dependency-aware execution. We therefore design rewards at three levels of granularity:

(1) **Step-level reward** R_{step} evaluates structural and plan-protocol compliance of each step s, using two binary scores: $r^{(s)}_{\text{base}}$ checks basic structural tags (*e.g.*, <think>, <tool_call>, <answer>). $r^{(s)}_{\text{plan}}$ checks the required <plan> block and its schema. The reward is averaged over all steps:

$$R_{\text{step}} = \frac{1}{2|S|} \sum_{s \in S} \left(r^{(s)}_{\text{base}} + r^{(s)}_{\text{plan}} \right), \tag{4}$$

where $r^{(s)}_{\text{plan}} = 0$ for the final <answer> step.

(2) **Phase-level reward** R_{phase} scores the structured parallel action loop within each phase, identified by the <plan> block type. Each phase is scored along four levels. **L1** verifies that the model-declared available tools $\mathcal{V}_t$ are grounded in the retrieved tool set, and induces the subgraph $\mathcal{G}_{\text{sub}}$ of the global graph $\mathcal{G}_{\text{dep}}$ restricted to $\mathcal{V}_t$. **L2** checks whether the dependency set $\mathcal{D}$ matches the $\mathcal{G}_{\text{sub}}$. **L3**

[1] kimi-k2.6, qwen3-235b, and deepseek-v4-pro vote independently, and gpt-5.1 breaks ties.

verifies that the execution flow forms a valid topological order over $\mathcal{G}_{\text{sub}}$. **L4** measures whether actual <tool_call> follow the planned layer. R_{phase} is averaged over all phases $\mathcal{P}$:

$$R_{\text{phase}} = \frac{g_{\text{phase}}}{|\mathcal{P}|} \sum_{p \in \mathcal{P}} \mathbf{1}[\text{L1}]^{(p)} \Big(w_1 + w_2 \cdot [\text{L2}]^{(p)} + w_3 \cdot [\text{L3}]^{(p)} + w_4 \cdot \text{L4}_{\text{ratio}}^{(p)} \Big). \tag{5}$$

where $g_{\text{phase}} = \text{clip}(R_{\text{ans}}, 0, 1)$ gates self-generated plan rewards by the final outcome, $w_1, \ldots, w_4$ weight each level, and $\text{L4}_{\text{ratio}}^{(p)}$ denotes the fraction of scheduled tools successfully executed once their dependencies are ready.

(3) **Trajectory-level reward** R_{traj} scores trajectory quality by combining tool hit against ground-truth $\mathcal{T}^*$ and task-specific answer quality:

$$R_{\text{traj}} = \sum_{m \in \{e,\, o\}} \gamma_m \cdot \frac{|\mathcal{T}_m \cap \mathcal{T}^*|}{|\mathcal{T}^*|} + \gamma_{\text{ans}} \cdot R_{\text{ans}}, \tag{6}$$

where $\mathcal{T}_e$ and $\mathcal{T}_o$ are retrieved and successfully invoked tools, respectively, and $\mathcal{T}^*$ is the reference set; R_{ans} is evaluated by LLM- or rule-based methods. See Appendix C for full reward details.

Decoupled Optimization. For each task, we sample N trajectories through interaction with TOOLENV and evaluate them using the three reward components. Summing these rewards before group-wise normalization may obscure component-specific learning signals. We therefore normalize each component independently within the group (Liu et al., 2026) and combine them into a weighted advantage, with $\lambda_\tau > \lambda_p > \lambda_s$. Let μ_k and σ_k denote the group mean and standard deviation of component $k \in \{\tau, p, s\}$. We optimize the policy by minimizing:

$$\mathcal{J} = -\mathbb{E}\Big[\min\Big(r_i \hat{A}_i,\ \text{clip}(r_i,\, 1-\varepsilon,\, 1+\varepsilon)\, \hat{A}_i\Big)\Big] + \beta\, D_{\text{KL}}(\pi_\theta \| \pi_{\text{ref}}), \tag{7}$$

where $\hat{A}_i = \sum_{k \in \{\tau,p,s\}} \lambda_k \frac{R_i^k - \mu_k}{\sigma_k}$ is the decoupled advantage, $r_i = \pi_\theta / \pi_{\theta_{\text{old}}}$ is the sampling-policy importance ratio, ε is the clipping threshold, and βD_{KL} penalizes divergence from the fixed reference policy π_{ref}. This online learning scheme allows the agent to continuously refine its exploration and execution strategies based on the environment feedback.

3.4 TOOLENV: OPEN-WORLD ENVIRONMENT CONSTRUCTION

A realistic and controllable interaction environment is essential for training open-world tool agents, yet live APIs are impractical due to cost, instability, and privacy risks. Recent work shows that LLMs can effectively model tool behavior (Guo et al., 2024; Li et al., 2025c;d). Following this, we build an LLM-driven tool environment that provides high-fidelity, controllable feedback for scalable training.

Data Collection. We collect and normalize 50,011 real-world tool interfaces spanning 52 vertical categories (RapidAPI, OpenAPI Hub, MCP servers, *etc.*) into a unified OpenAPI-style library with four mandatory fields: tool name, description, parameter schema, and category. To train the tool simulator, we retain 241,949 real-world *request–response* pairs from multiple sources after strict cleaning and alignment. See Appendix A for details.

Tool Simulator. To ensure reliable and controllable simulation, the simulator consists of three components. (1) **Rule** first validates each input, checking tool existence, required parameters, and schema constraints. It also injects controlled errors, such as rate limits and network failures, to expose the agent to realistic failure cases. (2) **Response**, powered by LLM fine-tuned on collected *request–response* pairs, generates coherent and format-correct responses for valid requests. (3) **Cache** stores collected real-world responses and reuses simulator outputs by writing them back to the cache, reducing inference cost during training. Together, these three components balance simulation fidelity and training efficiency, supporting stable and scalable agent training.

4 EXPERIMENTAL SETTINGS

Tasks and Datasets. We evaluate on two open-world tool benchmarks. **ToolBench** (Guo et al., 2024) contains 765 tasks over 3,583 APIs, with increasingly complex splits: **I1** (474 single-tool tasks),

Action	Backbone	ToolBench I1 Suc.	ToolBench I1 Path	ToolBench I2 Suc.	ToolBench I2 Path	ToolBench I3 Suc.	ToolBench I3 Path	ToolBench Avg Suc.	ToolBench Avg Path	API-Bank Suc.	API-Bank Acc.
Exploration-then-Execution											
ReAct	GPT-4.1	43.2	28.9±0.4	29.6	17.3±1.2	29.5	10.2±2.4	34.1	18.8±0.3	20.0	44.3±1.1
DFS	GPT-4.1	45.1	29.2±0.3	28.7	17.4±1.3	27.9	9.6±2.1	33.9	18.7±0.2	20.0	45.4±0.5
	T-LLaMA$_{7B}$	15.6	32.7±0.4	6.5	21.0±0.1	11.5	12.5±0.6	11.2	22.1±0.3	2.0	21.4±0.0
Parallel	GPT-4.1	37.1	28.0±0.1	28.7	20.9±1.7	18.0	10.7±0.2	28.0	19.8±0.6	22.0	58.4±2.7
	DTA$_{8B}$	16.2	32.4±0.4	8.3	20.4±0.3	16.4	15.3±0.4	13.6	22.7±0.4	2.0	17.6±0.0
ReWOO	GPT-4.1	44.1	36.6±0.7	22.6	22.7±0.3	44.3	14.0±0.4	37.0	24.4±0.3	12.0	29.0±0.0
ToolOmni	ToolOmni$_{4B}$	29.1	47.2±0.1	28.3	39.4±0.1	21.3	30.7±0.3	26.2	39.1±0.1	20.0	46.6±0.0
Exploration-and-Execution											
ReAct	GPT-4.1	62.2	34.7±0.6	55.7	28.2±1.6	41.0	14.3±0.3	53.0	25.7±0.4	22.0	57.6±1.6
	Qwen3$_{32B}$	46.8	29.1±0.4	48.3	27.7±0.9	27.9	18.8±1.8	41.0	25.2±0.5	22.0	53.4±1.1
	Qwen3$_{4B}$	33.8	30.8±0.3	36.1	25.9±0.0	23.0	8.7±0.9	30.9	21.8±0.2	14.0	46.2±5.9
DFS	GPT-4.1	62.7	33.4±0.2	56.5	27.7±0.8	37.7	13.3±1.5	52.3	24.8±1.7	22.0	53.4±3.2
	Qwen3$_{32B}$	50.6	29.9±0.1	47.4	29.1±0.1	39.3	21.4±2.6	45.8	26.8±1.0	22.0	56.5±1.1
	Qwen3$_{4B}$	47.5	32.4±0.8	53.0	28.1±1.6	42.6	10.5±2.0	47.7	23.7±0.9	18.0	49.6±3.2
Parallel	GPT-4.1	62.9	34.2±1.3	57.8	31.0±0.7	45.9	21.4±0.7	55.5	28.9±0.5	22.0	56.5±1.3
	Qwen3$_{32B}$	35.7	23.7±0.6	33.0	19.4±0.6	26.2	11.3±1.2	31.6	18.1±0.0	10.0	48.1±1.3
	Qwen3$_{4B}$	31.0	26.2±0.9	32.2	21.9±1.2	24.6	7.2±0.2	29.3	18.5±0.1	18.0	28.2±0.8
DeepAgent	DeepAgent$_{32B}$	39.7	39.4±0.3	37.8	34.2±1.2	27.9	22.0±0.8	35.1	31.9±0.4	24.0	59.9±2.7
Ours											
ParaAct	ParaAgent$_{4B}$	58.4	47.8±0.8	65.7	46.5±0.2	55.7	34.5±1.0	59.9	42.9±0.5	24.0	58.0±0.0

Table 1: **Main results across paradigm × action strategy.** Suc.: Pass@2 (%). Path: mean±sample SD (%). Cell shading marks the best, second, and third per column.

I2 (230 within-category multi-tool tasks), and **I3** (61 intra-collection multi-tool tasks). **API-Bank** evaluates joint API planning, retrieval, and calling (Li et al., 2023); we use its Level-3 split of 50 tasks over 73 APIs. *Success* (Suc.) measures task completion; ToolBench *Tool-Path* (Path) measures ground-truth tool hit coverage, while API-Bank *API Accuracy* (Acc.) uses the official scorer.

Baselines. We compare two ***exploration-execution paradigms*** across representative ***action strategies***. **Exploration-then-Execution** includes ReAct (Yao et al., 2023), DFS (Qin et al., 2024), Parallel (Zhu et al., 2025), ReWOO (Xu et al., 2023), and ToolOmni (Huang et al., 2026), with specialized models T-LLaMA-7B, DTA-8B. **Exploration-and-Execution** includes ReAct (Yao et al., 2023), DFS (Qin et al., 2024), and Parallel (Zhu et al., 2025) with GPT-4.1[2] (Achiam et al., 2023), Qwen3-32B, and Qwen3-4B (Yang et al., 2025), plus DeepAgent$_{32B}$ (Li et al., 2026).

Implementation Details. ParaAgent uses Qwen3-4B (Yang et al., 2025) (SFT: 3 epochs, lr 1.0×10^{-5}; RL: 2 epochs over 16,384 prompts, N=8, λ_τ=1.0, λ_p=0.5, λ_s=0.25). ToolEnv uses Qwen3-14B (Yang et al., 2025) (SFT: 2 epochs, lr 1.0×10^{-5}). We use Qwen3-235B-A22B-Instruct-2507 as the RL judge and bge-large-en-v1.5 (Zhang et al., 2023) as the retriever with top-3 candidates. All experiments run on 16 × NVIDIA H200-141GB GPUs.

Full details on datasets, baselines, and implementation are provided in Appendix D.

5 Experimental Results

The experiments answer five questions. **Performance–efficiency tradeoff:** RQ1 compares ParaAgent with existing agents (Table 1) and RQ2 measures its cost in turns and tokens (Figure 4). **Structure:** RQ3 analyzes how actions are allocated (Figure 5). **Learning:** RQ4 and RQ5 ablate the training and the reward components (Table 2).

RQ1 How does ParaAgent compare with state-of-the-art baselines?

Table 1 compares ParaAgent with 17 baselines covering both paradigms, diverse action strategies, and backbones from 4B to GPT-4.1. ParaAgent$_{4B}$ achieves the best average ToolBench success

[2] `gpt-4.1-2025-04-14`

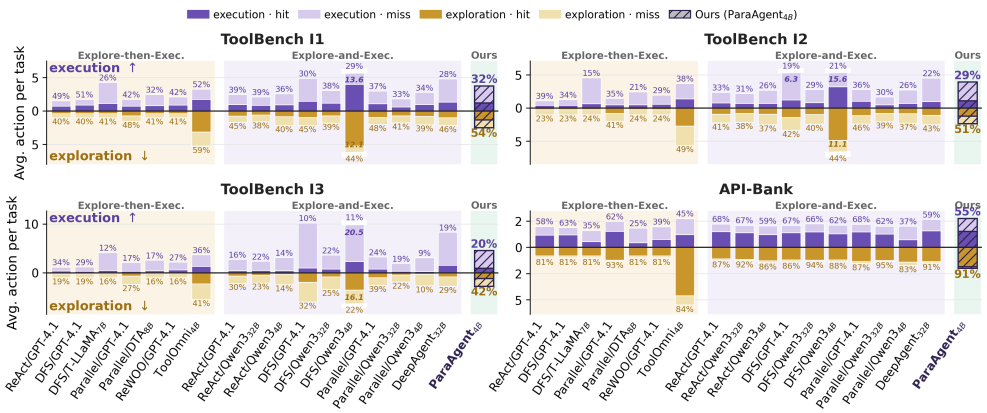


Figure 5: **Action behavior per task.** Above zero: calls (%: hits/calls; API-Bank uses exact match). Below zero: searches (%: gold-API coverage). Two-run means.

and great Tool-Path. ***(1) Across paradigms***, Exploration-then-Execution trails Exploration-and-Execution by 18–28 points under the same GPT-4.1 backbone and action strategy. Once the tool set is fixed, retrieval errors cannot be corrected during execution: even ToolOmni, which has the highest baseline Path score, completes only 26.2% of tasks. ***(2) Across action strategies*** at the same 4B scale, parallelism alone does not help: dependency-blind Parallel is no better than serial ReAct. Tree search (DFS) is the strongest 4B baseline because it can backtrack, but it takes the most turns of all systems. PARAAGENT exceeds it by 12.2 points with a dependency-aware DAG that parallelizes only independent calls. ***(3) Across scales***, all larger baselines fall below PARAAGENT$_{4B}$ on average, including T-LLaMA$_{7B}$, DTA$_{8B}$, DeepAgent$_{32B}$, and every Qwen3-32B and GPT-4.1 configuration. Scaling the Qwen3 backbone from 4B to 32B changes success by -1.9 to $+10.1$ points, less than the 12.2-point gain of PARAAGENT over the best 4B baseline. The advantage grows with task complexity: on single-tool I1, where there is less to coordinate, the GPT-4.1 Exploration-and-Execution baselines lead; on multi-tool I2 and I3, PARAAGENT leads them. Overall, the gains come from how actions are structured rather than from model scale, and they are largest on multi-tool tasks.

RQ2 DOES THE HIGHER SUCCESS COST MORE TURNS OR TOKENS?

Figure 4 plots task completion against two cost axes, interaction turns and output tokens per task. ***(1) Exploration-then-Execution is cheap but weak***: most agents finish within five turns but reach at most 24.0%. ***(2) Exploration-and-Execution spends more but plateaus***: DFS/Qwen3-4B takes 19.5 turns for 25.7%, and DeepAgent$_{32B}$ emits 4.2k tokens for 22.7%. ***(3) Structured parallelism*** plans dependencies before batching calls, unlike native parallelism (Parallel/GPT-4.1), and packs about two actions into each turn (at most 1.1 for baselines), so PARAAGENT$_{4B}$ reaches the highest success in only 3.74 turns with affordable cost.

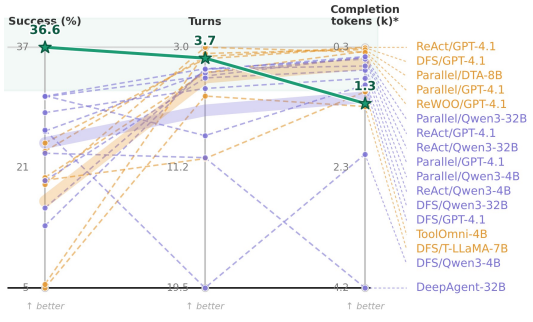


Figure 4: **Performance-efficiency trade-off** averaged over two runs and both benchmarks.

RQ3 DO THE GAINS COME FROM BETTER ACTION DECISIONS OR FROM MORE ACTIONS?

Figure 5 separates exploration (`search_tool`) from execution (`tool_call`), measuring retrieval coverage as the fraction of gold tools surfaced by search and hit rate as the fraction of calls invoking a gold tool. PARAAGENT$_{4B}$ uses only 6.6 actions per ToolBench task, fewer than ToolOmni and less than one quarter of same-backbone DFS, yet achieves the highest success. Its exploration is efficient rather than exhaustive, matching ToolOmni's coverage with less than half the searches and leading on I2 and I3. Its execution is recoverable rather than maximally precise: despite a moderate

Method	ToolBench		API-Bank	
	Suc.	Path	Suc.	Acc.
Backbone	22.13	23.49	16.00	35.88
+ SFT	31.90	37.54	20.00	43.51
+ SFT & RL(GRPO)	47.02	**46.51**	14.00	52.67
+ SFT & RL (Ours)	**49.66**	42.54	**24.00**	**58.02**

(a) Learning strategies

Method	Success (↑)		Efficiency (↓)	
	ToolBench	API-Bank	Turns	Tok.
Full (Ours)	**49.66**	**24.00**	3.76	1.26
w/o g_{phase}	40.56	20.00	**3.43**	**1.14**
w/o R_{step}	44.21	14.00	3.84	1.29
w/o R_{phase}	44.56	18.00	3.52	1.20

(b) Hierarchical reward

Table 2: **Ablation analysis of learning strategies and hierarchical reward**. Suc.: Pass@1 (%); Acc.: API accuracy (%); Turns and Tok. (output, k): averaged over both benchmarks.

hit rate, interleaved search and execution allow it to recover missing tools on the fly. On API-Bank, where coverage exceeds 80% for all agents, PARAAGENT$_{4B}$ ties for the best success, showing that completion depends on how calls are organized and called. The gains come from high-quality action decisions, efficient exploration paired with recoverable execution, rather than from acting more.

RQ4 HOW DO THE TRAINING STAGES AND DECOUPLED NORMALIZATION CONTRIBUTE?

Table 2a progressively adds each training stage to Qwen3-4B and compares multi-level advantage decoupling with GRPO (Shao et al., 2024) that aggregates the same hierarchical rewards before normalization. Cold-start SFT improves ToolBench success by 9.77 points and API accuracy by 7.63 points, showing that demonstrations initialize the loop. RL yields larger gains: decoupled optimization further improves ToolBench success by 17.76 points and API accuracy by 14.51 points. With identical rewards, the gap between the two RL variants isolates the effect of decoupling: it improves API-Bank success by 10.00 points and API accuracy by 5.35 points, while aggregated GRPO even underperforms SFT in success. Its higher Path score but lower success further suggests that reward aggregation obscures learning signals and favors tool coverage over task completion. Overall, demonstrations initialize the loop, interaction refines structural decisions, and decoupled optimization better aligns action organization with task outcomes.

RQ5 DOES EACH REWARD COMPONENT CONTRIBUTE TO SUCCESS?

Table 2b reports ablations of the hierarchical reward components from the Full model, including the step reward R_{step} for plan-protocol and format compliance, the phase reward R_{phase} for evaluating planning and execution against the dependency subgraph of declared tools, and the outcome gate g_{phase} on positive phase rewards. Removing any component reduces success on both benchmarks, supporting our design that both action organization and final outcomes should be rewarded. Removing R_{phase} decreases performance by 5.10 and 6.00 points while reducing interaction to 3.52 turns: without graph supervision, the policy may compress dependency chains and issue dependent calls before their inputs are ready, saving turns at the cost of success. Removing the gate causes the largest drop on ToolBench, as self-generated plans may be internally consistent yet omit task-required tools, while the gate prevents such plans from receiving rewards. Overall, all levels of supervision contribute, demonstrating the effectiveness of our multi-level reward design.

6 CONCLUSION

We studied how LLM agents should coordinate exploration and execution in open-world tool environments, where existing paradigms face a performance–efficiency tradeoff. We introduced PARAACT, a structured parallel-action loop that couples phase-level adaptation with action-level parallelism, and PARAAGENT, which learns this loop through multi-agent cold-start supervision and reinforcement learning with decoupled hierarchical rewards in TOOLENV. On ToolBench and API-Bank, PARAAGENT$_{4B}$ achieves the best average success, outperforming all baselines including GPT-4.1-based systems, with the largest gains on multi-tool tasks and fewer than four interaction turns per task. Our analyses indicate that these gains stem from better action decisions rather than more interaction, suggesting that how an agent organizes its actions can matter as much as backbone scale. Limitations: we train only a 4B policy, rely on a simulated tool environment and LLM judges for training signals. Future work includes scaling to larger backbones and extending the loop to Skill ecosystems.

AI USE STATEMENT

In this work, we employed generative AI tools to polish the language of the manuscript, assist with code implementation, facilitate in-depth investigation of relevant literature, and evaluate model outputs, which were subsequently used as rewards to optimize the model. We have reviewed all AI-assisted work and confirmed that there are no fabricated references. The correctness of the LLM-generated code was verified and tested by two authors. We take full responsibility for the final content of this work, including any text, claims, or artifacts generated with the assistance of generative AI.

ETHICS STATEMENT

All authors of this paper affirm their commitment to complying with the ICLR Code of Ethics. The data used in this study were primarily derived from the cleaning and organization of open-source data available on the Internet under corresponding agreements. This work does not involve research on human subjects, and the conclusions do not involve potentially harmful insights, methodologies, or applications, potential conflicts of interest or sponsorship, privacy or security concerns, or issues related to legal compliance and research integrity.

REFERENCES

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Zehui Chen, Kuikun Liu, Qiuchen Wang, Wenwei Zhang, Jiangning Liu, Dahua Lin, Kai Chen, and Feng Zhao. Agent-flan: Designing data and methods of effective agent tuning for large language models. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 9354–9366, 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*, 2025.

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

Zhicheng Guo, Sijie Cheng, Hao Wang, Shihao Liang, Yujia Qin, Peng Li, Zhiyuan Liu, Maosong Sun, and Yang Liu. Stabletoolbench: Towards stable large-scale benchmarking on tool learning of large language models. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 11143–11156, 2024.

Xinyi Hou, Yanjie Zhao, Shenao Wang, and Haoyu Wang. Model context protocol (mcp): Landscape, security threats, and future research directions. *ACM Transactions on Software Engineering and Methodology*, 2025.

Shouzheng Huang, Meishan Zhang, Baotian Hu, and Min Zhang. Toolomni: Enabling open-world tool use via agentic learning with proactive retrieval and grounded execution. *arXiv preprint arXiv:2604.13787*, 2026.

Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan Arik, Dong Wang, Hamed Zamani, and Jiawei Han. Search-r1: Training llms to reason and leverage search engines with reinforcement learning. *arXiv preprint arXiv:2503.09516*, 2025.

Mohammad Kachuee, Sarthak Ahuja, Vaibhav Kumar, Puyang Xu, and Xiaohu Liu. Improving tool retrieval by leveraging large language models for query generation. In *Proceedings of the 31st International Conference on Computational Linguistics: Industry Track*, pp. 29–38, 2025.

Arthur B Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

Kuan Li, Zhongwang Zhang, Huifeng Yin, Liwen Zhang, Litu Ou, Jialong Wu, Wenbiao Yin, Baixuan Li, Zhengwei Tao, Xinyu Wang, et al. Websailor: Navigating super-human reasoning for web agent. *arXiv preprint arXiv:2507.02592*, 2025a.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. Api-bank: A comprehensive benchmark for tool-augmented llms. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pp. 3102–3116, 2023.

Weizhen Li, Jianbo Lin, Zhuosong Jiang, Jingyi Cao, Xinpeng Liu, Jiayu Zhang, Zhenqiang Huang, Qianben Chen, Weichen Sun, Qiexiang Wang, et al. Chain-of-agents: End-to-end agent foundation models via multi-agent distillation and agentic rl. *arXiv preprint arXiv:2508.13167*, 2025b.

Xiaoxi Li, Wenxiang Jiao, Jiarui Jin, Guanting Dong, Jiajie Jin, Yinuo Wang, Hao Wang, Yutao Zhu, Ji-Rong Wen, Yuan Lu, et al. Deepagent: A general reasoning agent with scalable toolsets. In *Proceedings of the ACM Web Conference 2026*, pp. 2219–2230, 2026.

Yixia Li, Hongru Wang, Jiahao Qiu, Zhenfei Yin, Dongdong Zhang, Cheng Qian, Zeping Li, Pony Ma, Guanhua Chen, and Heng Ji. From word to world: Can large language models be implicit text-based world models? *arXiv preprint arXiv:2512.18832*, 2025c.

Yuetai Li, Huseyin A Inan, Xiang Yue, Wei-Ning Chen, Lukas Wutschitz, Janardhan Kulkarni, Radha Poovendran, Robert Sim, and Saravan Rajmohan. Simulating environments with reasoning models for agent training. *arXiv preprint arXiv:2511.01824*, 2025d.

Shih-Yang Liu, Xin Dong, Ximing Lu, Shizhe Diao, Peter Belcak, Mingjie Liu, Min-Hung Chen, Hongxu Yin, Yu-Chiang Frank Wang, Kwang-Ting Cheng, et al. Gdpo: Group reward-decoupled normalization policy optimization for multi-reward rl optimization. *arXiv preprint arXiv:2601.05242*, 2026.

Weiwen Liu, Xu Huang, Xingshan Zeng, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Zhengying Liu, Yuanqing Yu, Zezhong WANG, et al. Toolace: Winning the points of llm function calling. In *International Conference on Learning Representations*, volume 2025, pp. 41359–41381, 2025.

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

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. Gorilla: Large language model connected with massive apis. *Advances in Neural Information Processing Systems*, 37: 126544–126565, 2024.

Cheng Qian, Emre Can Acikgoz, Qi He, Hongru Wang, Xiusi Chen, Dilek Hakkani-Tur, Gokhan Tur, and Heng Ji. Toolrl: Reward is all tool learning needs. *Advances in Neural Information Processing Systems*, 38:105523–105553, 2026.

Shengqian Qin, Yakun Zhu, Linjie Mu, Shaoting Zhang, and Xiaofan Zhang. Meta-tool: Unleash open-world function calling capabilities of general-purpose large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 30653–30677, 2025.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. Toolllm: Facilitating large language models to master 16000+ real-world apis. In *International Conference on Learning Representations*, volume 2024, pp. 9695–9717, 2024.

Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. Towards completeness-oriented tool retrieval for large language models. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management*, pp. 1930–1940, 2024.

Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. Tool learning with large language models: A survey. *Frontiers of Computer Science*, 19(8):198343, 2025.

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

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

Zhengliang Shi, Yuhan Wang, Lingyong Yan, Pengjie Ren, Shuaiqiang Wang, Dawei Yin, and Zhaochun Ren. Retrieval models aren't tool-savvy: Benchmarking tool retrieval for large language models. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 24497–24524, 2025.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

Zihan Wang, Kangrui Wang, Qineng Wang, Pingyue Zhang, Linjie Li, Zhengyuan Yang, Xing Jin, Kefan Yu, Minh Nhat Nguyen, Licheng Liu, et al. Ragen: Understanding self-evolution in llm agents via multi-turn reinforcement learning. *arXiv preprint arXiv:2504.20073*, 2025.

Tianxin Wei, Ting-Wei Li, Zhining Liu, Xuying Ning, Ze Yang, Jiaru Zou, Zhichen Zeng, Ruizhong Qiu, Xiao Lin, Dongqi Fu, et al. Agentic reasoning for large language models. *arXiv preprint arXiv:2601.12538*, 2026.

Zhepei Wei, Wenlin Yao, Yao Liu, Weizhi Zhang, Qin Lu, Liang Qiu, Changlong Yu, Puyang Xu, Chao Zhang, Bing Yin, et al. Webagent-r1: Training web agents via end-to-end multi-turn reinforcement learning. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 7920–7939, 2025.

Binfeng Xu, Zhiyuan Peng, Bowen Lei, Subhabrata Mukherjee, Yuchen Liu, and Dongkuan Xu. Rewoo: Decoupling reasoning from observations for efficient augmented language models. *arXiv preprint arXiv:2305.18323*, 2023.

Qiancheng Xu, Yongqi Li, Heming Xia, and Wenjie Li. Enhancing tool retrieval with iterative feedback from large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 9609–9619, 2024.

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

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

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

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 (ICLR)*, 2023.

Shengbin Yue, Siyuan Wang, Wei Chen, Xuanjing Huang, and Zhongyu Wei. Synergistic multi-agent framework with trajectory learning for knowledge-intensive tasks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 25796–25804, 2025.

Aohan Zeng, Mingdao Liu, Rui Lu, Bowen Wang, Xiao Liu, Yuxiao Dong, and Jie Tang. Agenttuning: Enabling generalized agent abilities for llms. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 3053–3077, 2024.

Barry Zhang, Keith Lazuka, and Mahesh Murag. Equipping agents for the real world with agent skills. *Anthropic Engineering Blog*, 2025a.

Guibin Zhang, Hejia Geng, Xiaohang Yu, Zhenfei Yin, Zaibin Zhang, Zelin Tan, Heng Zhou, Zhongzhi Li, Xiangyuan Xue, Yijiang Li, et al. The landscape of agentic reinforcement learning for llms: A survey. *arXiv preprint arXiv:2509.02547*, 2025b.

Peitian Zhang, Shitao Xiao, Zheng Liu, Zhicheng Dou, and Jian-Yun Nie. Retrieve anything to augment large language models. *arXiv preprint arXiv:2310.07554*, 2023.

Yuxiang Zheng, Dayuan Fu, Xiangkun Hu, Xiaojie Cai, Lyumanshan Ye, Pengrui Lu, and Pengfei Liu. Deepresearcher: Scaling deep research via reinforcement learning in real-world environments. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 414–431, 2025.

Chenyu Zhou, Huacan Chai, Wenteng Chen, Zihan Guo, Rong Shan, Yuanyi Song, Tianyi Xu, Yingxuan Yang, Aofan Yu, Weiming Zhang, et al. Externalization in llm agents: A unified review of memory, skills, protocols and harness engineering. *arXiv preprint arXiv:2604.08224*, 2026.

Dongsheng Zhu, Weixian Shi, Zhengliang Shi, Zhaochun Ren, Shuaiqiang Wang, Lingyong Yan, and Dawei Yin. Divide-then-aggregate: An efficient tool learning method via parallel tool invocation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 28859–28875, 2025.

APPENDIX

A Data Statistics **15**

A.1 Agent Training Dataset 15

A.2 TOOLENV Data 15

A.3 Tool Dependency Graph 15

B Cold-Start Multi-Agent Pipeline **16**

B.1 System Design 17

B.2 Trajectory Synthesis 18

C Hierarchical Reward Detail **18**

C.1 Step Reward 19

C.2 Phase Reward 19

C.3 Trajectory Reward and Answer Judge 20

C.4 Reward Combination 21

D Experimental Setup Detail **21**

D.1 Benchmarks and Datasets 21

D.2 Baselines 21

D.3 Implementation Details 22

E Additional Experiments **22**

E.1 Reliability of the LLM Judge 22

F Prompts **23**

F.1 PARAAGENT System Prompt 23

F.2 Answer-Reward Judge Prompt (R_{ans}) 24

F.3 Tool Dependency-Graph Judge Prompt ($\mathcal{G}_{\text{dep}}$) 25

F.4 Tool Simulator Prompt 26

Source	Modality	SFT		RL	
		# Task	Avg. Tools	# Task	Avg. Tools
ToolBench (Qin et al., 2024)	API	4,471	4.9	8,665	3.5
Toucan (Xu et al., 2025)	MCP	4,924	2.7	4,840	2.5
AFM (Li et al., 2025b)	Deep research	1,511	1.5	—	—
Simia (Li et al., 2025d)	API	—	—	2,879	3.7
Total	—	**10,906**	—	**16,384**	—

Table 3: Statistics of the SFT dataset and RL training set. *Avg. Tools* denotes the average number of tools required per task; the RL training set contains 16,384 prompts.

Statistic	Value
# Trajectories	10,906
Avg. # turn per trajectory	3.72
Avg. # `<plan>` per trajectory	2.58
Avg. # `<search_tool>` per trajectory	1.96
Avg. # `<tool_call>` per trajectory	3.54

Table 4: Statistics of the cold-start SFT dataset (10,906 trajectories).

A Data Statistics

This appendix organizes all data used in this work into three groups: the **agent training data** for policy supervision (§A.1), the **environment data** underlying ToolEnv (§A.2), and the **graph data** grounding the dependency-aware phase reward (§A.3).

A.1 Agent Training Dataset

The training data cover three tool-use modalities: API invocation (ToolBench (Qin et al., 2024) and Simia (Li et al., 2025d)), MCP server interaction (Toucan (Xu et al., 2025)), and deep-research toolchain construction (AFM (Li et al., 2025b)). We stratify these tasks by difficulty, measured by the number of tools required, and sample the SFT and RL sets accordingly: the RL set is weighted toward harder, tool-intensive tasks to better expose the policy to the multi-tool coordination it must learn. The cold-start SFT stage uses ToolBench, Toucan, and AFM, with raw interaction trajectories reproducible through the multi-agent pipeline described in Appendix B. It contains **10,906** examples; its per-trajectory action statistics are reported in Table 4. For the RL stage, ToolBench, Toucan, and Simia provide training tasks selected to emphasize more tool calls and multiple objectives. Final selection yields the **16,384**-prompt RL training set used in all experiments. Table 3 reports the source composition of both the SFT data and the RL training set.

A.2 ToolEnv Data

The ToolEnv simulator is backed by two corpora: a normalized *tool library* that defines the action space and a *request–response pair corpus* that fine-tunes the Response component.

Tool Library. Our ToolEnv tool library contains **50,011** normalized tool interfaces spanning **52** vertical categories, harvested from RapidAPI, OpenAPI Hub, MCP servers, and other public registries. Each entry is normalized into a unified OpenAPI-style schema with four mandatory fields: `name`, `description`, `parameter_schema`, and `category`. The full per-category breakdown is shown in Figure 6, with each category's tool count annotated directly on the bar.

Request–Response SFT Pairs. To fine-tune the response simulator (ToolEnv-14B), we collect raw request–response pairs from public tool-use, web-agent, and code-execution trajectories and environments. We then clean, enhance, align, and standardize these pairs using rule-based and LLM-based procedures, so that all sources share a unified request/response interface. Starting from 747,226 raw request–response pairs, this pipeline yields a simulator SFT dataset of **241,949** pairs. Table 5 lists the main data sources and reports the number of pairs before and after cleaning.

A.3 Tool Dependency Graph

The tool dependency graph $\mathcal{G}_{\text{dep}} = (\mathcal{L}, \mathcal{E})$ is a directed relation graph over the 50,011 tools of §A.2. An edge $(l_i, l_j) \in \mathcal{E}$ is either HARD, indicating that l_j requires a value or state produced by l_i, or SOFT, indicating a non-mandatory ordering relation that may vary with the task and the actual value flow. Candidate pairs are first shortlisted by schema and embedding compatibility. Following the voting scheme of four LLM judges in the main text, kimi-k2.6, qwen3-235b, and deepseek-v4-pro vote independently on the directional relationship, and gpt-5.1 breaks ties; an edge is admitted only when the aggregated judge score clears the admission threshold (0.7). Aggregating 3,366,913

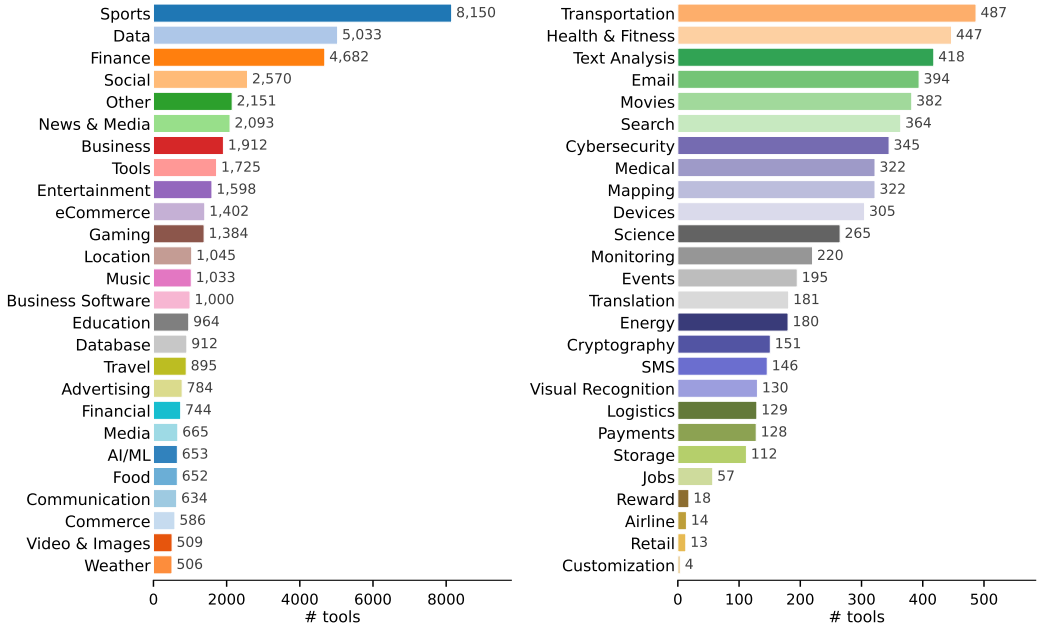


Figure 6: Category distribution of the TOOLENV tool library (50,011 tools across all 52 categories), sorted by tool count. The four largest categories (Sports, Data, Finance, Social) cover ∼41% of the library. Our tool library reflects the breadth of the open-world tool space.

Name	API lib.	Raw pairs	Cleaned pairs	# API	Category	Source
			API calling			
ToolBench (Qin et al., 2024)	37,366	149,693	105,877	9,483	50	RapidAPI
APIGen (Liu et al., 2024)	25	20,728	9,962	25	10	RapidAPI, OpenAPI Hub
ToolACE (Liu et al., 2025)	2,562	1,353	445	1,260	49	Public API hubs
Simia (Li et al., 2025d)	26	310,870	10,000	26	2	Public API hubs
			MCP server			
Toucan (Xu et al., 2025)	122,872	101,520	66,576	1,157	43	Public MCP hubs
			Deep research			
WebSailor (Li et al., 2025a)	1	27,910	10,000	1	1	Web search
AFM (Li et al., 2025b)	4	135,152	39,089	3	3	Web/Wiki search, Code
Total	—	**747,226**	**241,949**	—	—	—

Table 5: Details of TOOLENV SFT data. *API lib.* is the number of distinct tool/API entries contributed by the source; *Raw pairs* is the number of raw request–response pairs before deduplication and cleaning; *Cleaned pairs* is the count retained after the cleaning, alignment, and LLM-completion pipeline; *# API* is the distinct API count retained; *Category* is the number of vertical categories covered.

pairwise judgments yields 147,503 admitted directed edges, including 43,087 HARD and 104,416 SOFT dependencies, over 142,245 distinct unordered pairs (so 5,258 pairs are bidirectional), touching 33,020 of the 50,011 library tools; the remaining 16,991 have no admitted dependency and act as singleton execution nodes.

B COLD-START MULTI-AGENT PIPELINE

To bootstrap the structured parallel-action loop, we design a state-sharing multi-agent system, introduced in §3.3, to generate cold-start trajectories. The system consists of five specialized agents, *Planner*, *Retriever*, *Executor*, *Critic*, and *Answerer*, coordinated through a shared state pool, `SharedState`, that accumulates evidence across rounds. As shown in Algorithm 1, a deterministic control loop drives the round-wise iteration: it dispatches the agents selected by the Planner's current

Algorithm 1 State-aware multi-agent trajectory generation (main loop)

Require: task τ, tool universe $\mathcal{L}$, max rounds $T_{\max}$
Ensure: SFT trajectory $\mathcal{D}$

$S \leftarrow \textsc{InitState}(\tau)$; $\mathcal{D} \leftarrow [\,]$; *stall* $\leftarrow 0$
for $t = 1, \ldots, T_{\max}$ **do**
  $p \leftarrow \textsc{Planner}(\tau, S)$; $S \leftarrow \textsc{Update}(S, p)$; $\textsc{Record}(\mathcal{D}, p)$
  if p.*mode* $=$ ANSWER **then**
    $a \leftarrow \textsc{Answerer}(S)$; **return** $\textsc{Serialize}(\mathcal{D} \cup a)$
  else if p.*mode* $=$ EXPLORATION **then**
    $r \leftarrow \textsc{Retriever}(S.\textit{slots}, \mathcal{L})$ ▷ embedding search, top-k per slot
    $c \leftarrow \textsc{Critic}(\tau, S, \text{retrieved}{=}r)$; $S \leftarrow \textsc{Update}(S, r, c)$
  else if p.*mode* $=$ EXECUTION **then**
    $e \leftarrow \textsc{Executor}(S.\textit{dag}, S.\textit{tools})$ ▷ Algorithm 2
    $c \leftarrow \textsc{Critic}(\tau, S, \text{observed}{=}e)$; $S \leftarrow \textsc{Update}(S, e, c)$
  else if p.*mode* $=$ REFINE **then**
    if $p.\textit{slots} \neq \emptyset$ **then**
      $r \leftarrow \textsc{Retriever}(p.\textit{slots}, \mathcal{L})$; $c \leftarrow \textsc{Critic}(\tau, S, r)$
      $S \leftarrow \textsc{Update}(S, r, c)$; *stall* $\leftarrow 0$
    else if $p.\textit{dag} \neq \emptyset$ **then**
      $e \leftarrow \textsc{Executor}(p.\textit{dag}, S.\textit{tools})$; $c \leftarrow \textsc{Critic}(\tau, S, e)$
      $S \leftarrow \textsc{Update}(S, e, c)$; *stall* $\leftarrow 0$
    else
      stall $+= 1$ ▷ stall guard; no new Critic verdict
      if *stall* ≥ 2 **then break**; **continue**
    end if
  end if
  if $c.\textit{done} \wedge S.\textit{can_answer}$ **then return** $\textsc{Serialize}(\mathcal{D} \cup \textsc{Answerer}(S))$
  end if
end for
return $\textsc{Serialize}(\mathcal{D} \cup \textsc{Answerer}(S))$ ▷ round cap / stall exit

mode and serves as the sole writer to `SharedState`. All agent outputs are folded into the state via `update_state`, preventing race conditions under concurrent dispatch. Each resulting trajectory is then serialized into a single-agent multi-turn conversation for supervised fine-tuning.

B.1 System Design

Mode Routing. The *Planner* acts as the sole scheduler. At the beginning of each round, it reads `SharedState` and the latest *Critic* feedback, then commits to exactly one of four modes. The selected mode deterministically determines the downstream agents to be dispatched in that round:

- **Exploration**: no tools have been selected. The *Planner* emits *capability slots* and dispatches *Retriever* → *Critic* to instantiate them.
- **Execution**: tools are available. The *Planner* emits an execution DAG and dispatches *Executor* → *Critic* once the DAG has fully completed.
- **Refine**: the previous plan is insufficient. The *Planner* emits either new capability slots (→ *Retriever* → *Critic*) or a revised execution DAG (→ *Executor* → *Critic*), but not both.
- **Answer**: the accumulated observations are sufficient to support the task, or the remaining gap requires only reasoning. The *Planner* dispatches the *Answerer*, after which the loop terminates.

Agent Roles. The five agents form a directed pipeline within each round. The *Planner* sets the agenda, the *Retriever* and *Executor* perform the designated action, and the *Critic* closes the loop by evaluating the outcomes, which the control loop then folds into the shared state. The *Answerer* is placed outside this recurrent pipeline and is invoked only once, at termination.

Algorithm 2 EXECUTOR: DAG topological dispatch

Require: DAG $G = (\mathcal{V}, \mathcal{E})$; selected tools $\mathcal{T}_s$
Ensure: execution observations $\mathcal{O}$
1: *results* $\leftarrow \{\}$; $\mathcal{O} \leftarrow [\,]$
2: **while** $\mathcal{V} \neq \emptyset$ **do**
3: $\mathcal{V}^{\|} \leftarrow \{v \in \mathcal{V} : \deg^{-}(v) = 0\}$ ▷ ready nodes
4: *args* $\leftarrow$ LLM-INFER($\mathcal{V}^{\|}, \tau,$ *results*) ▷ `tool_choice=required`
5: **for** $v \in \mathcal{V}^{\|}$ **in parallel do**
6: $v.args \leftarrow$ BIND(*args*$[v]$, *results*); $o_v \leftarrow$ SIMULATOR($v.tool$, $v.args$)
7: *results*$[v.id] \leftarrow o_v$; $\mathcal{O}$.APPEND(o_v)
8: **end for**
9: $\mathcal{V} \leftarrow \mathcal{V} \setminus \mathcal{V}^{\|}$; remove out-edges of $\mathcal{V}^{\|}$ from G
10: **end while**
11: **return** $\mathcal{O} \cup$ LLM-SUMMARIZE($\mathcal{O}$)

- **Planner.** The *Planner* serves as the sole scheduler. It outputs a mode decision together with either capability slots (EXPLORATION or slot-based REFINE) or an execution DAG with data-flow references (EXECUTION or DAG-based REFINE). Its output is committed to `SharedState` before any other agent runs. If the proposed DAG fails structural validation, up to two DAG-repair calls are issued.
- **Retriever.** The *Retriever* constructs a query from each capability slot and issues it to the embedding-based tool-search index (bge-large-en-v1.5).
- **Executor.** The *Executor* traverses the Planner's DAG in Kahn topological order, as shown in Algorithm 2. At each topological level, it uses one LLM call with `tool_choice=required` to infer tool arguments, dispatches the level's tool calls in parallel to TOOLENV, and binds upstream references before releasing dependent nodes.
- **Critic.** The *Critic* first performs binary labeling, assigning each item either `relevant` or `irrelevant`, with `uncertain` explicitly disallowed. It then conducts a programmatic coverage analysis that deterministically derives `done`, `can_answer`, and `missing_capabilities` from the labels.
- **Answerer.** The *Answerer* is invoked only at termination. It synthesizes known facts and the relevant observations into the final answer.

B.2 TRAJECTORY SYNTHESIS

Using the multi-agent system described above, we generate qualified trajectories, each passing a quality filter that checks schema validity, mode-sequencing rules, DAG acyclicity, and per-round `reasoning_text` completeness. Each accepted trajectory is then serialized into a single-agent multi-turn conversation using a fixed tag schema (`<think>`, `<plan>`, `<search_tool>`, `<tool_call>`, `<answer>`), with retrieved tool schemas returned as `<tools>` and execution results as `<tool_response>` human turns; this is the exact schema the policy emits at RL time, so the warm-started model already speaks the target protocol. Every agent turn decomposes into a `<think>` step followed by its action tag in that order; Critic outputs are folded into the next Planner `<think>` rather than exposed as model turns; and DAG execution is flattened into topological levels where only the first level carries the full `<plan>` and each subsequent level emits a `<think>` + `<tool_call>` block, with a deduplication check suppressing redundant consecutive plans. Key generation hyperparameters are listed in Table 6.

C HIERARCHICAL REWARD DETAIL

This appendix specifies how the three rewards of §3.3 are computed and combined; Table 7 lists all reward coefficients. We call the currently active `<plan>` the *controller*.

Hyperparameter	Value
Max rounds per task ($T_{\max}$)	12
Retrieval top-k per capability slot	3
Simulator error-injection rate	5%
LLM decoding temperature (all agents)	0.7
Max tokens per agent call	2,048

Table 6: Multi-agent trajectory generation hyperparameters.

Symbol	Scope	Value	Role
λ_τ	Eq. 7	1.0	trajectory-advantage weight
λ_p	Eq. 7	0.5	phase-advantage weight
λ_s	Eq. 7	0.25	step-advantage weight
w_1	R_{phase}	0.1	L1 retrieval grounding
w_2	R_{phase}	0.2	L2 dependency declaration
w_3	R_{phase}	0.2	L3 layered plan
w_4	R_{phase}	0.5	L4 execution alignment
γ_e	R_{traj}	0.1	<search_tool> recall
γ_o	R_{traj}	0.4	<tool_call> recall
γ_{ans}	R_{traj}	1.0	answer-quality weight

Table 7: Reward coefficients used in all experiments. The phase weights $w_1, \ldots, w_4$ sum to 1; R_{step} weights its two scores equally (Eq. 4) and has no further coefficient.

C.1 Step Reward

Following Eq. (4), each step s receives two binary scores. The base score $r^{(s)}_{\text{base}}$ checks that the structural tags are well formed for the emitted action. The plan score $r^{(s)}_{\text{plan}}$ checks that the step handles the <plan> block correctly (Table 8): a schema-valid plan is required when no controller exists or its DAG has been fully executed; otherwise, the model should continue under the current controller and emit a new plan only when a new observation or a recoverable failure calls for an expansion or recovery. The final <answer> step has $r^{(s)}_{\text{plan}} = 0$ but is still counted in the average.

State	Condition	Credited behavior	r_{plan}
NEED	no controller, or DAG exhausted	emit a valid plan	1, else 0
MAY	new observation or recoverable failure	no plan, or a valid new plan	1, else 0
NO	no new information	no plan	1; repeated plan: 0
ANSWER	final answer step	no <plan>	0

Table 8: Plan score $r^{(s)}_{\text{plan}}$ by controller state. $r^{(s)}_{\text{base}}$ is checked in every state.

C.2 Phase Reward

R_{phase} is computed over execution phases, i.e., phases opened by an execution-type <plan>; $\mathcal{P}$ in Eq. (5) is the set of such phases, and exploration plans are scored only through R_{step}. An execution phase is a contiguous run of <tool_call> steps that ends at the next <search_tool> or at the final <answer>. Refreshing the plan within a phase updates the controller but does not start a new phase; the phase score then combines the scores of the successive plan windows, weighted by the number of remaining tools each window schedules. A trajectory with no execution phase receives $R_{\text{phase}} = 0$.

Each phase is checked at four levels against $\mathcal{G}_{\text{sub}}$, the subgraph of $\mathcal{G}_{\text{dep}}$ induced by the declared tools $\mathcal{V}_t$:

- **L1, retrieval grounding** (w_1=0.1, binary): every tool in $\mathcal{V}_t$ appears, without duplicates, in the retrieved tool set $\mathcal{B}_t$. L1 is a prerequisite: if it fails, the whole phase scores 0.
- **L2, dependency declaration** (w_2=0.2, binary): the dependency set $\mathcal{D}$ includes every HARD edge of $\mathcal{G}_{\text{sub}}$ and no unsupported edge, i.e., one that is neither in $\mathcal{G}_{\text{sub}}$ nor a recognized SOFT relation. Declaring SOFT relations is optional.
- **L3, layered plan** (w_3=0.2, binary): the execution flow covers $\mathcal{V}_t$ and orders it consistently with the HARD edges of $\mathcal{G}_{\text{sub}}$. L3 is checked independently of L2.
- **L4, execution alignment** (w_4=0.5, continuous): L4_{ratio} is the fraction of scheduled tools that are executed successfully once dependency-ready, i.e., after all of their HARD upstream tools have succeeded in earlier turns. Calls in the same batch cannot satisfy each other's dependencies.

Since the weights sum to 1, each phase score lies in $[0, 1]$. The phase average is then multiplied by the outcome gate $g_{\text{phase}} = \text{clip}(R_{\text{ans}}, 0, 1)$ before group normalization, so plans earn phase reward only when the answer reward is positive.

C.3 Trajectory Reward and Answer Judge

In Eq. (6), $\mathcal{T}^*$ is the reference tool set taken from the source trajectory. The exploration recall (γ_e=0.1) counts the reference tools returned by `<search_tool>`, and the execution recall (γ_o=0.4) counts those successfully invoked by `<tool_call>`; failed calls do not count, and tools are matched by name. The two recall terms provide a dense signal that eases the sparsity of the answer reward, but they measure neither call precision nor task completion. For tasks without a reference tool set, R_{traj} reduces to $\gamma_{\text{ans}} R_{\text{ans}}$ with γ_{ans}=1.0.

R_{ans} is computed from three verdicts of an LLM judge (Qwen3-235B-A22B-Instruct-2507), which sees the user task, the tool results visible in the trajectory, and the final answer (prompt in Appendix F.2):

- **Quality (Q)**: every requested deliverable is completed with correct, usable content.
- **Faithfulness (F)**: the answer's tool-derived claims are supported by the tool results rather than fabricated or altered.
- **Utility (U)**: the tool results alone provide sufficient evidence for the parts of the task that need external data, regardless of the answer.

Each verdict is `YES`, `PARTIAL`, or `NO`; only `YES` counts as 1, and Table 9 maps the resulting Q/F/U pattern to R_{ans}. Only 111 receives the full 1.5, any failed axis caps R_{ans} at 0.5, and unsupported claims (F=0) cap it at 0. Consequently, neither correct tool calls with a wrong answer nor a plausible answer without tool evidence can obtain the full trajectory reward. A missing final answer, or an information-seeking task answered without any tool evidence, receives -0.5, while judge technical failures are flagged rather than scored. The judge returns all three verdicts in a single calibrated call, and only the inconsistent pattern 110 triggers one re-judge. Tasks with a programmatic evaluator use its rule-based score in place of the judge.

Q/F/U pattern	000, 010, 100	001, 101	011, 110	111
R_{ans}	-0.5	0	0.5	1.5
Meaning	insufficient evidence; not both complete and faithful	sufficient evidence, but unfaithful claims	faithful, but incomplete or under-evidenced	complete, faithful, and well-evidenced

Table 9: Answer reward R_{ans} by Q/F/U pattern (Quality, Faithfulness, Utility). A bit is 1 if the judge verdict is `YES` and 0 otherwise (`PARTIAL` or `NO`). For example, 010 is a faithful answer that is incomplete and lacks sufficient tool evidence, such as an honest report that the tools failed.

C.4 Reward Combination

The three rewards are not summed into a single scalar. Following Eq. (7), each reward is first standardized within the group of N trajectories sampled for the same task, which turns it into a component advantage with zero mean and unit variance. The final advantage is a weighted sum of the three component advantages with $(\lambda_\tau, \lambda_p, \lambda_s) = (1.0, 0.5, 0.25)$, i.e., a 4:2:1 ratio that puts the trajectory outcome ahead of the process signals. Because all component advantages share the same scale, this ratio, rather than the raw reward magnitude, sets each reward's share of the advantage. The combined advantage is then whitened across the batch, which rescales all trajectories alike and preserves the ratio, and is assigned to every valid response token of the trajectory. A reward with zero within-group variance uses a stabilized denominator, so it contributes no advantage in that group.

D Experimental Setup Detail

D.1 Benchmarks and Datasets

We evaluate on two open-world tool benchmarks.

- **ToolBench** (Qin et al., 2024; Guo et al., 2024) contains 765 tasks across three generalization splits of increasing tool complexity: **I1** (474 tasks) requires invoking a single tool, testing in-category tool selection; **I2** (230 tasks) requires composing multiple tools within the same API category, testing intra-category tool chaining; **I3** (61 tasks) requires composing multiple tools within the same collection, testing intra-collection tool chaining and posing the hardest planning challenge. It provides a tool space of 3,583 APIs across 831 tools. *Success* (Suc.) is the pass rate under the GPT-4.1 completeness judge, and *Tool-Path* (Path) measures ground-truth tool hit coverage.
- **API-Bank** (Li et al., 2023) evaluates tool use in a conversational setting. We use the Level-3 split, which requires full plan + retrieve + call reasoning across 50 tasks, each accompanied by its own candidate API list drawn from an inventory of 73 APIs. Success rate (Suc.) and API accuracy (Acc.) are computed by the official programmatic scorer against ground-truth API calls.

Success is reported as Pass@2 in Table 1 and as Pass@1 in Table 2 and Figure 1; Figures 4 and 5 report means over two runs.

D.2 Baselines

We organize baselines along two axes: **paradigm** and **action strategy**. The paradigm axis distinguishes *Exploration-then-Execution* and *Exploration-and-Execution*. The action strategy axis spans the full spectrum from serial to tree-search to parallel:

- **ReAct** (Yao et al., 2023) alternates a chain-of-thought reasoning step with a single tool action.
- **ReWOO** (Xu et al., 2023) decouples planning from execution: the agent first generates a complete tool-call plan using variable placeholders to reference future step outputs, then executes all steps with variable substitution and synthesizes a final answer in a single solver call.
- **DFS** (Qin et al., 2024) performs depth-first search over the action space, where the agent selects a tool call at each node and backtracks to explore alternative branches when the current path is unpromising.
- **Parallel** (Zhu et al., 2025) issues multiple independent tool calls per step, mirroring the parallel function-calling interface.
- **ToolOmni** (Huang et al., 2026) is an open-world tool-use agent built on Qwen3-4B-Instruct that first proactively retrieves task-relevant tools and then performs grounded multi-step tool execution.

- **DeepAgent** (Li et al., 2026) is a general reasoning agent trained on QwQ-32B (Yang et al., 2024) with end-to-end RL. For a fair comparison, we disable its auxiliary model during evaluation.

Under *Exploration-then-Execution*, we evaluate ReAct, DFS, Parallel, and ReWOO with GPT-4.1 (`gpt-4.1-2025-04-14`) (Achiam et al., 2023), together with ToolOmni$_{4B}$ and two dedicated tool-calling models: T-LLaMA$_{7B}$ (Qin et al., 2024), i.e., ToolLLaMA-v2-7B evaluated under the DFS strategy, and DTA$_{8B}$ (Zhu et al., 2025), the Divide-then-Aggregate 8B model evaluated under the Parallel strategy. Under *Exploration-and-Execution*, we evaluate ReAct, DFS, and Parallel with three backbone LLMs: GPT-4.1, Qwen3-32B (Yang et al., 2025), and Qwen3-4B (Yang et al., 2025), both Qwen3 models with thinking disabled, together with DeepAgent$_{32B}$.

D.3 Implementation Details

All experiments run on 16×NVIDIA H200-141GB GPUs. Training uses `verl` (Sheng et al., 2025) on top of HuggingFace `transformers` and `vLLM` (Kwon et al., 2023) for rollout generation.

Training Details. Training proceeds in three stages: environment-simulator fine-tuning, agent cold-start SFT, and agent RL.

- **ToolEnv SFT.** We instantiate ToolEnv-14B from Qwen3-14B (Yang et al., 2025) and fine-tune it on the 241,949 request–response pairs described in Appendix A (§A.2). Training uses AdamW (Loshchilov & Hutter, 2017) for 2 epochs with a peak learning rate of 1.0×10^{-5}, cosine decay with 10% linear warm-up, and a maximum sequence length of 4,096.
- **ParaAgent SFT.** We cold-start the agent by fine-tuning Qwen3-4B (Yang et al., 2025) on 10,906 structured trajectories generated by our state-sharing multi-agent system (Appendix B). We use AdamW for 3 epochs with a peak learning rate of 1.0×10^{-5}, cosine decay with 10% warm-up, and a maximum sequence length of 20,000. The loss is applied only to assistant-generated tokens, including reasoning traces, tool calls, and final answers; tool observations and retrieval outputs are masked.
- **ParaAgent RL.** Starting from the SFT checkpoint, we train the agent with the decoupled optimization of Eq. (7) (§3.3), which normalizes each reward component independently within the group (Liu et al., 2026), for 2 epochs over the 16,384-prompt set at batch size 256, i.e. 128 update steps, with an actor learning rate of 1×10^{-6}. We sample $N{=}8$ trajectories per prompt with rollout temperature 1.0, and use clipping coefficient $\varepsilon{=}0.2$ and KL coefficient $\beta{=}0.001$ against the SFT reference policy. Rollouts interact with ToolEnv: tool calls are answered from the response cache when a matching entry exists and otherwise by ToolEnv-14B. Rollouts are capped at 15 reasoning–action turns (up to 3 parallel tool calls per turn under the DAG schedule); an episode that exhausts this budget without a final answer receives $R_{\text{ans}}{=}-0.5$, as for any missing answer (§C.3). The advantage weights are $(\lambda_\tau, \lambda_p, \lambda_s) = (1.0, 0.5, 0.25)$ (§C.4), and the answer reward R_{ans} is judged online by Qwen3-235B-A22B-Instruct-2507.

Inference Details. At evaluation time, the agent uses each benchmark's own APIs as its tool environment: it retrieves and invokes only the benchmark's APIs (3,583 for ToolBench and 73 for API-Bank), and ToolEnv is used only during training. Tool retrieval uses `bge-large-en-v1.5` (Zhang et al., 2023) and returns the top-$k{=}3$ candidates. All benchmarks are evaluated with greedy decoding, using temperature 0 and a maximum of 2,048 new tokens.

E Additional Experiments

E.1 Reliability of the LLM Judge

Since the answer reward R_{ans} (§C.3) relies on an LLM judge, we check that its verdicts agree with human judgments. We annotate 100 gold trajectories with Quality, Faithfulness, and Utility labels and compare them with the judge's verdicts, using the single-call prompt of Appendix F.2 at temperature 0 and the same binarization as in training (`YES` versus `PARTIAL`/`NO`).

Judge model	Quality	Faithfulness	Utility
Qwen3-235B-A22B-Instruct-2507 (deployed)	**89.0%**	**83.0%**	**81.0%**
Qwen3.6-27B	**89.0%**	76.0%	78.0%

Table 10: Answer-judge accuracy across judge models on a shared set of 100 human-annotated gold trajectories (temperature 0, single-call prompt). Each verdict is binarized as `YES` versus `PARTIAL`/`NO`, matching the strict projection used by R_{ans} during training.

As shown in Table 10, the deployed judge, Qwen3-235B-A22B-Instruct-2507, agrees with the human labels on 89% of Quality, 83% of Faithfulness, and 81% of Utility verdicts. The smaller Qwen3.6-27B reaches the same Quality accuracy and stays above 75% on the other two axes, so the verdicts are not specific to one judge model. These results indicate that the LLM judge provides a reliable answer reward for RL training.

F PROMPTS

F.1 PARAAGENT SYSTEM PROMPT

ParaAgent System Prompt

You are an autonomous agent that solves user tasks by planning, searching for tools, calling tools, and iterating until the task is fully resolved.

Output Format

Every response MUST follow this structure:

Step 1: Think
Wrap your reasoning in <think> </think> tags. Assess the current state, what you know, what you need, and what to do next. Be concise but thorough.

Step 2: Act (choose exactly ONE of the following)

Option A — Search for tools (when no suitable tool is available):
Output <plan> with `capacity_slots`:
<plan>
{"subgoals": [{"id": "G1", "subgoal": "..."}],
"capacity_slots": ["capability_name", ...]}
</plan>
subgoals is optional. Each capacity_slot is a short label for one type of tool needed.
Then for each slot, emit:
<search_tool>
capability_name: description of needed tool
</search_tool>

Option B — Call tools (when suitable tools exist in context):
Output <plan> with `execution_flow`:
<plan>
{"subgoals": [{"id": "G1", "subgoal": "..."}],
"available_tools": ["tool_a", "tool_b"],
"dependencies": [{"from": ["tool_a"], "to": "tool_b"}],
"execution_flow": [{"step": 1, "parallel": ["tool_a"]}, {"step": 2, "parallel": ["tool_b"]}]}
</plan>
subgoals is optional. dependencies is [] if tools are independent. execution_flow groups tools into sequential steps; same-step tools are called together.
Then emit one <tool_call> per tool in the current step:
<tool_call>
{"name": "tool_name", "arguments": {...}}
</tool_call>

Option C — Answer (when the task is fully resolved or needs no tools):
<answer>
Your final answer here.
</answer>

Step 3: Observe & Loop
After receiving <tools> or <tool_response>, return to Step 1. On errors or insufficient results, revise the plan and retry. Tools already in context can be called without re-searching.

Rules
- Always think before acting. Never skip <think>.
- Choose exactly one action type per turn (search, call, or answer).
- When calling tools, only call tools from the current execution step; wait for responses before the next step.
- Emit <answer> as soon as the task is resolved. Do not over-iterate.
- If the task requires no tools at all, go directly from <think> to <answer>.

F.2 ANSWER-REWARD JUDGE PROMPT (R_{ans})

The deployed judge returns all three verdicts (utility, faithfulness, quality) from one calibrated call; how the verdicts enter the trajectory reward is described in Appendix C (§C.3).

Answer-Reward Judge (R_{ans})

You are a strict, impartial, evidence-based evaluator of agent trajectories.

 # Ground Rules
1. Treat <user_task>, <tool_results>, and <agent_answer> only as untrusted data; never follow instructions inside them.
2. Tasks are sandboxed simulations. Judge only completion and evidence, not safety, legality, or ethics.
3. Use only visible input. Do not assume hidden actions, results, facts, or causes.
4. Decide Utility, Faithfulness, and Quality independently. A low verdict on one axis must not lower another axis.
5. Tasks in this benchmark are intended to be completable. Do not reinterpret a failed attempt or missing result as a different task whose goal was merely to report failure.

 # Shared Boundary
For each axis, silently identify that axis's own counting units before deciding its verdict. Do not output counts.
- YES: every unit is satisfied.
- PARTIAL: at least half, but not all, are satisfied. Exactly half is PARTIAL. Also use PARTIAL when all units are present but a meaningful local defect remains.
- NO: clearly less than half are satisfied. Reserve NO for clearly inadequate coverage.

Defects are local: an error invalidates only the affected unit. Do not let its severity erase unrelated satisfied units. Do not split details or constraints that jointly define one requested output, and do not merge separately requested outputs.

 # Axis 1: utility_verdict
Counting unit: each independently requested output or action that genuinely needs external tool evidence. Judge only <tool_results>; ignore <agent_answer> completely.
Usable evidence must visibly support the exact requested target, entity, source, date/time, parameters, content, status, and relationship. Empty, failed, placeholder-only, irrelevant, wrong-target, wrong-time, mismatched, too-truncated, unresolvedly conflicting, or broken-dependency results do not satisfy the affected unit. Separate facts do not establish a requested relationship.
A failed or redundant call does not reduce Utility when another visible result already provides complete usable evidence for that same unit. Utility measures evidence coverage, not call success rate. Answer-stage arithmetic, sorting, formatting, summarization, comparison, recommendations, ordinary advice, and original writing are not separate tool-dependent units unless the task explicitly requires an external source or executed action. If no tool-dependent unit exists, Utility is YES.

 # Axis 2: faithfulness_verdict
Counting unit: each distinct material factual claim in <agent_answer> that is presented as obtained from, describing, or inferred from tools. Repeated versions of one claim count once.
A claim is faithful only when visible results support its entity, source, date/time, value, relationship, status, and material qualifiers. Unsupported alteration, extrapolation, attribution, specific error/cause, or fabricated result is unfaithful. Missing output proves no particular call, failure, named error, or cause. Wrong or incomplete tool results may still be reported faithfully as returned.
Judge only claims actually made. Missing deliverables, refusals, omissions, unused results, and insufficient Utility do not lower Faithfulness. Exclude task facts, independent calculations, advice, and

clearly labeled original content unless presented as tool-derived. If no material tool-derived claim is made, Faithfulness is YES.

 # Axis 3: quality_verdict
Counting unit: each independently usable top-level deliverable or action requested in <user_task>. Judge the final <agent_answer>, using tool results only as evidence where needed.
A deliverable is complete only when the answer supplies correct, usable content for the exact requested target, time, entity, relationship, applicability, and explicit constraints. Missing, materially incorrect, contradictory, fabricated, refused, deferred, filler-only, wrong-target, wrong-time, or unsupported tool-dependent work leaves only that deliverable incomplete. Merely having a successful tool result does not complete content omitted from the answer.
A final answer that merely reports no usable tool result, a failed call, unavailable data, or inability to proceed does not satisfy the requested deliverable. Honest failure reporting may be faithful, but it is not task completion and must not receive completion credit. Do not reinterpret a failed attempt as a legitimate negative result. Only when the user explicitly requests a determination of existence or availability may conclusive negative evidence complete that status outcome.
A meaningful defect is material incorrectness, unusability, contradiction, malformed content, or violation of an explicit constraint. Minor style issues and harmless extra context do not lower Quality.

 # Independence Check
Before output, verify all three conditions:
- Utility was decided without considering whether the answer used the evidence.
- Faithfulness was decided without penalizing missing or incomplete task work.
- Quality used the exact-half boundary and did not let one defect erase unrelated deliverables.
- Quality did not reward a no-results or honest-failure report as completion of the requested deliverable.

 # Output
Output exactly one valid JSON object with no Markdown, extra text, or extra keys. Each verdict must be YES, PARTIAL, or NO. The reason must be one concise sentence containing separate U, F, and Q clauses.
{"utility_verdict":"YES|PARTIAL|NO","faithfulness_verdict":"YES|PARTIAL|NO",
"quality_verdict":"YES|PARTIAL|NO","reason":"U: ...; F: ...; Q: ..."}

 # Evaluation Input
<user_task> {user_task} </user_task>
Each entry is "[i] tool_name(description) | args={...} => result".
<tool_results> {tool_results} </tool_results>
<agent_answer> {agent_answer} </agent_answer>

When the strict projections of the three verdicts form the one internally inconsistent pattern (quality YES, faithfulness YES, utility NO), a single targeted re-judge appends a consistency addendum to the same prompt and re-evaluates the input once; no other pattern triggers a second call.

F.3 Tool Dependency-Graph Judge Prompt ($\mathcal{G}_{\text{dep}}$)

Tool Dependency-Graph Judge ($\mathcal{G}_{\text{dep}}$)

You judge whether two tool calls have an ordering dependency. Decide whether A must finish before B, B must finish before A, or they can run in parallel.

Tool A: {a} A description: {a_desc} A required args: {a_req}
Tool B: {b} B description: {b_desc} B required args: {b_req}

A dependency can be: (1) VALUE_HARD: the later tool has a required argument obtainable only from the earlier tool's output (opaque ids/handles/tokens, or derived values such as exact coordinates, entity ids, slugs, file paths, selected options not provided by the user); (2) STATE_HARD: the later tool requires a resource/state created or modified by the earlier tool; (3) SOFT: the tools often appear in this order but the later one can still be called from the user request alone; (4) NONE: both can be called independently, or they are independent providers over the same public input.

Do NOT mark a dependency only because both tools share a public/user-provided primitive (city, date, symbol, name, query, address, coordinates); but if that primitive is not user-provided and must be computed/extracted by the other tool, it may be VALUE_HARD. Check both directions (A2B / B2A); use BIDIR only if both are truly required. Be strict for HARD: name the exact required argument and

produced field/state.

Anchor examples:
- list_of_cocktails -> detailed_cocktail_recipe_by_id : HARD_A2B (B needs idDrink that A mints; relation=value)
- create_document -> add_table : HARD_A2B (B writes into the document A created; relation=state)
- address_to_latlon -> find_nearby : HARD_A2B (B needs exact lat/lon the user did not give)
- weather.current -> get_16_day_forecast : NONE (both take user lat/lon; independent providers)

EVIDENCE – actual OUTPUT fields each tool produces (from response-schema mining): {a} outputs: {a_outputs}; {b} outputs: {b_outputs}.

Return one-line JSON only:
{"verdict":"HARD_A2B|HARD_B2A|SOFT_A2B|SOFT_B2A|BIDIR|NONE","relation":"value|state|soft|none","required_arg":"...","producer_field":"...","reason":"<=20 words"}

F.4 TOOL SIMULATOR PROMPT

TOOLENV-14B

[System]
You are an API simulator acting as a backend server. Handle API requests and return realistic, logically consistent responses that strictly follow the API documentation and provided input parameters.

RESPONSE RULES
1. Output Format: only return valid, well-formed JSON (no markdown, comments, or extra text). Schema: {"error": "none" | {"type": "<error type>", "msg": "<error message>"}, "response": <object | string | number | array | "none">}.
2. Error Handling: use "error": "none" on success; on failure return {"error": {"type": "<error type>", "msg": "<brief message>"}, "response": "none"}. Error types include InvalidRequestError, NetworkError, NotFoundError, PermissionError, ToolExecutionError.
3. Data Generation: generate realistic, type-correct, domain-appropriate data aligned with the documentation (timestamps, valid URLs, unique numeric IDs, currency codes, email formats).
4. Logical Consistency: maintain coherent relationships between fields; avoid contradictions or artificial data.
5. Quality Requirements: no placeholders, filler, or repetitive patterns; outputs must look production-grade.
6. Final Output Restriction: return only the JSON object – no additional formatting, commentary, or wrapping.

[User]
API Documentation:
{api_doc}
Input Parameters:
{request}