SCALING TOOL-USE AGENTS VIA ENVIRONMENT-GROUNDED QUERY SYNTHESIS

Anonymous authors
Paper under double-blind review

ABSTRACT

Training tool-use agents with agentic reinforcement learning (Agentic RL) is bottlenecked by two challenges: the lack of scalable, robust execution environments and the scarcity of realistic training data that carries learning signal and captures implicit human reasoning. Existing pipelines build the two separately from pre-collected tool corpora and overlook the difficulty and realism of synthesized tasks, limiting their effectiveness for RL. We introduce **ENVFACTORY**, a fully automated framework that addresses both challenges. ENVFACTORY autonomously explores and verifies stateful, executable tool environments from authentic resources, and synthesizes natural multi-turn trajectories through topology-aware sampling and implicitness injection, producing grounded queries with realistic intents. Using 85 environments across 7 domains, ENVFACTORY generates 2,575 SFT and RL trajectories and beats prior work with fewer environments and less data. Across Qwen3 models from 1.7B to 8B, ENVFACTORY improves over the base models by up to **15.0 points** on BFCL-v3 multi-turn, **8.6** on MCP-Atlas, **4.9** on τ^2-Bench, and **8.3** on VitaBench. By automating both environment construction and trajectory synthesis, ENVFACTORY offers a scalable, extensible foundation for Agentic RL.

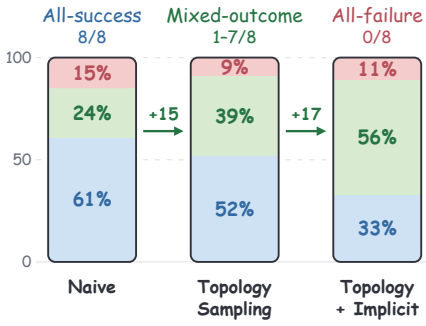
1 INTRODUCTION

Equipping Large Language Models (LLMs) with tools enables real-time information retrieval, precise computation, and interaction with external systems (Qu et al., 2025; Luo et al., 2025). Early work teaches tool-calling formats and interaction patterns through *supervised fine-tuning* (SFT) (Yang et al., 2025b; Xu et al., 2025); more recently, *agentic reinforcement learning* (Agentic RL) lets agents improve through trial-and-error interaction with executable environments (Hao et al., 2025; Jin et al., 2025; Feng et al., 2026). Scaling either paradigm depends on two ingredients: **environments** that execute reliably, and **training tasks** that carry learning signal. Existing pipelines build the two in separate stages, constructing environments from a pre-collected corpus and synthesizing tasks afterwards (Cai et al., 2025; Fang et al., 2025; Song et al., 2026). We argue that the two are coupled. An environment should be self-contained and dependency-complete, otherwise it's merely a toolset. Similarly, a task should be grounded in that environment, with every prerequisite reachable through its tools and state so that rollouts can succeed and be verified.

However, existing approaches fall short on both fronts. On the environment side, recent work scales code-generated tool-use environments along different axes. Stateless environments (Sullivan et al., 2025; Ye et al., 2026) are the easiest to scale, but because their tool calls neither read nor change a shared state, they cannot teach an agent how earlier actions constrain later ones. Other works build environments from a preassembled tool or task corpus (Cai et al., 2025; Fang et al., 2025; Song et al., 2026). Their tools are chosen to cover the corpus rather than to model a complete system, so an environment may lack the state or producer tools that its own tasks depend on, and it can only cover domains the corpus already includes.

A second gap lies on the data side. Our diagnostic study (Figure 1a) shows that directly prompting an LLM to synthesize tasks from an environment produces outcomes clustered at two extremes: 61% of tasks succeed on every rollout and 15% fail on every rollout. From the perspective of GRPO (Shao et al., 2024) and DAPO (Yu et al., 2025), such tasks carry little learning signal because they offer zero group advantage. We attribute the two extremes to different causes. All-failure tasks often have

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.



(a) Task outcomes

Method	Autonomous Discovery	Tool Graph	Dependency Resolution	Implicitness Injection
AgentScaler	✗	✓	✗	✗
AutoForge	✗	✓	✗	✓
EnvScaler	✗	✗	✗	✗
AWM	✗	✗	✗	✗
ScaleEnv	✗	✓	✗	✗
ENVFACTORY	✓	✓	✓	✓

(b) Method comparison

Figure 1: (a) Outcomes for 100 tasks per method over 8 rollouts each. Mixed-outcome tasks balance solvability and difficulty, indicating a higher-quality training distribution with nonzero binary-success variance for group-relative learning. (b) Mechanisms of related systems (Section 2). *Autonomous Discovery* uses documentation retrieved during construction; crosses mark components not reported by the cited work.

unresolved tool dependencies: they require inputs that neither the user query nor any earlier call can provide. All-success tasks are often over-specified: they spell out every action and identifier, reading like a robotic instruction list rather than a natural user request. We therefore target tasks that are **solvable** within the environment, **difficult** enough to challenge the current policy, and **realistic** in leaving inferable actions implicit (Wang et al., 2025; Xu et al., 2025).

We introduce **ENVFACTORY**, an automated framework that builds stateful environments and multi-turn training trajectories around this coupling, with a *dependency tool graph* as the hub between the two. For environments, ENVFACTORY discover candidate environment by retrieving online APIs and builds stateful tools through iterative testing and refinement to ensure the environment is correctly executable and self-contained. A parameter-level tool graph then links environments to tasks, and two mechanisms target the two outcome extremes. ***Topology-aware sampling*** resolves tool dependencies by recursively sampling the missing producers of each argument, so that every accepted tool is dependency-complete; this reduces the all-failure share of synthesized tasks from 15% to 9%. ***Implicitness injection*** then compresses inferable actions and replaces explicit identifiers with contextual references, reducing the all-success share from 52% to 33%. Together, the two mechanisms raise the share of mixed-outcome tasks from 24% to 56%. These shifts increases the share of tasks that provide a non-degenerate signal for group-relative learning (Figure 1a).

Using ENVFACTORY, we construct 85 verified environments with 842 tools across diverse domains, including *commerce, finance, travel, office, lifestyle, research, and utilities*, as illustrated in Figure 2. Building on these environments, we synthesize 1,622 SFT and 953 RL multi-turn, multi-step trajectories for post-training. Despite using fewer environments than concurrent work (Song et al., 2026; Wang et al., 2026), ENVFACTORY achieves higher training efficiency and stronger downstream performance, improving Qwen3-series models by up to 15.0 points on BFCL-v3 (Patil et al., 2025), 8.6 points on the real-world MCP benchmark MCP-Atlas (Bandi et al., 2026), 4.9 points on τ^2-Bench (Barres et al., 2025), and 8.3 points on VitaBench (He et al., 2026b).

2 RELATED WORK

Environment Scaling for Tool Agents. Tool-agent training environments fall into three families. **Production environments** expose real APIs (Qin et al., 2024) or MCP servers (Xu et al., 2025), but face authentication, availability, and latency constraints at scale. **LLM-simulated environments** model tool behavior and state (Li et al., 2025; 2026b; Chen et al., 2026), but may violate the intended state transitions (Kalai et al., 2025; Wang et al., 2024). **Code-generated environments** implement tools and databases as sandboxed executable code (Fang et al., 2025; Cai et al., 2025; Wang et al., 2026; Song et al., 2026; Hao et al., 2026; Tu et al., 2026); ENVFACTORY belongs to this family. Figure 1b compares five related systems across four mechanisms. For *Autonomous Discovery*, AutoForge and AgentScaler use pre-collected tool descriptions or API pools, EnvScaler mines existing task sets, AWM expands abstract domain seeds, and ScaleEnv synthesizes schemas

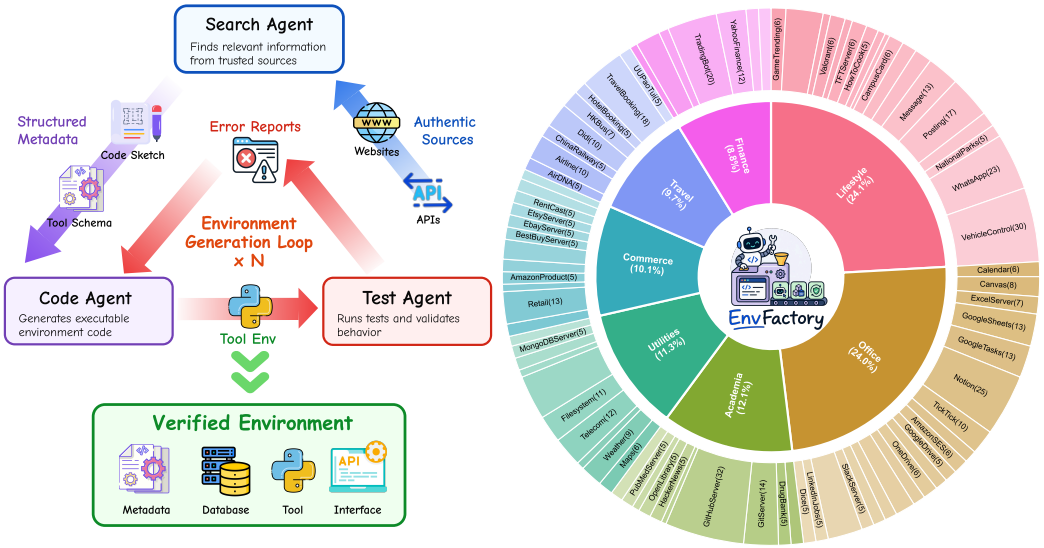


Figure 2: Left: *EnvGen* retrieves source documents, implements stateful tools, and tests and revises them. Right: domain proportions (inner ring) and tool counts per environment (outer ring).

from domain keywords. ENVFACTORY retrieves online API documentation and sample payloads during construction, giving interfaces and default values documented provenance. Its tests establish executable consistency with documented interfaces (Section 3.2), the target abstraction for scalable training environments.

Dependency Graphs and Task Synthesis. Multi-tool tasks involve dependencies (He et al., 2026a; Li et al., 2026a; Wu et al., 2025). Tool graphs (*Tool Graph*) derive edges from past trajectories (Chen et al., 2025), semantic relations among tool descriptions and parameters (Wang et al., 2025), or LLM-inferred workflows (Fang et al., 2025). EnvScaler instead generates tasks from a synthesized initial state, while AWM derives schemas from generated tasks; neither uses an explicit graph. Graph samplers generally expand forward: AgentScaler takes random directed walks; AutoForge merges walk paths and adds reasoning nodes; Magnet (Yin et al., 2025) edits signature paths; and ScaleEnv (Tu et al., 2026) extends a seed chain with tools whose inputs are already satisfiable. None inserts missing producers for a selected consumer. ENVFACTORY distinguishes parameter-flow from workflow edges and resolves each consumer's inputs backward, within a depth bound (*Dependency Resolution*, Section 3.3.2); unresolved chains are rejected. For requests, AutoForge favors minimal, natural descriptions and Magnet inserts implicit nested calls at the signature level. ENVFACTORY instead applies explicit refinement operations to state-grounded requests (*Implicitness Injection*, Section 3.4). Appendix F discusses further related work.

3 METHOD

3.1 PROBLEM SETUP

We consider an agent that serves a user by calling tools in stateful environments.

Environments and Tools. An environment is a tuple $e = (m, \mathcal{D}, \pi, \mathcal{V}_e)$, where m is its metadata (description and tool schemas), $\mathcal{D}$ is the database schema of its state, π is its executable implementation, and $\mathcal{V}_e$ is the set of tools it exposes through MCP (Anthropic, 2024). Given a set of environments $\mathcal{E}$, the agent can access the toolset $\mathcal{V} = \bigcup_{e \in \mathcal{E}} \mathcal{V}_e$. Each tool $v \in \mathcal{V}$ maps inputs in $\mathcal{I}(v)$ to outputs in $\mathcal{O}(v)$ and may read or modify the environment state s, an instance of $\mathcal{D}$.

Interaction. An episode starts from an initial state s_0 and a user request. At each step, the agent either calls a tool in $\mathcal{V}$ and observes its output, or replies to the user in natural language; the user may then answer questions, clarify, or give further instructions. The episode ends when the user emits a stop token or a step limit is reached, yielding a trajectory ξ of messages and tool calls and a terminal state s_T. We judge task success on both ξ and s_T (Section 3.5).

3.2 Environment Construction

Overview. *EnvGen* constructs a new environment $e_{\text{new}} = (m, \mathcal{D}, \pi, \mathcal{V}_{e_{\text{new}}})$ with optional user guidance by proposing candidates from authentic sources and iteratively implementing, executing, and revising them until verified (Figure 2). An accepted environment is added to the pool as $\mathcal{E} \leftarrow \mathcal{E} \cup \{e_{\text{new}}\}$.

Proposal and Sketch. A Search Agent analyzes the current pool $\mathcal{E}$ to identify coverage gaps and retrieves external sources—API documentation, technical reports, and usage examples—that describe broadly used functionality. For each selected candidate, it produces structured metadata m containing the description and schemas. To avoid dead ends, it organizes each environment from a lifecycle perspective: an attribute that can be increased must also be reducible, and a state that can be entered must provide a valid transition out. This lifecycle closure makes the specification self-contained and dependency-complete, while grounding it in documented interfaces provides provenance and preserves diversity as the pool grows.

Database Modeling and Code Implementation. Given m, a Code Agent derives a stateful database schema $\mathcal{D}$ that captures the required entities, relationships, and mutable state. Tool parameters, intermediate states, and persistent records are formalized as Pydantic schemas with standardized load and dump interfaces, supporting isolated and reproducible execution across rollouts. Conditioned on m and $\mathcal{D}$, the Code Agent implements executable Python code π for each tool, following the specified functionality, constraints, and state transitions, and exposes it through a standardized MCP interface $\mathcal{V}_{e_{\text{new}}}$.

Revision Loop. A Test Agent generates unit tests and validates the environment against five criteria: (1) tool interfaces match m; (2) tools import and execute; (3) results match the expected behavior; (4) database state transitions correctly after each call; and (5) attribute and tool lifecycles are complete and mutually consistent. A candidate whose functionality cannot be coherently supported by the available tools is returned for revision. On each failure, the Test Agent produces a structured report, and the Code Agent updates the affected component and rebuilds the environment. The loop repeats until all execution and completeness checks pass. If the revision budget is exhausted, the candidate is discarded.

The verified tools and state transitions provide the substrate for the two task-synthesis mechanisms that follow. Topology-aware sampling composes them into solvable plans, while implicitness injection turns those plans into realistic requests without exposing every required action.

3.3 Dependency Tool Graph

3.3.1 Tool Graph Construction

To guide trajectory synthesis, we construct a typed dependency graph $G = (\mathcal{V}, E_{\text{flow}}, E_{\text{wf}})$ over the global toolset, whose nodes are tools. A *parameter-flow edge* $(v_i \xrightarrow{p_i} v_j) \in E_{\text{flow}}$ records that an output of v_i can supply the specific input parameter $p_i \in \mathcal{I}(v_j)$; a *workflow edge* $(v_i \to v_j) \in E_{\text{wf}}$ records only that v_j plausibly follows v_i. Backward dependency resolution uses E_{flow} alone, whereas forward expansion of a plan may follow either type. The graph is built in two steps.

Step 1: Semantic Parameter Matching. Using the `BAAI/bge-m3` embedding model (Chen et al., 2024), we encode the name and description of every input and output parameter of every tool. For each ordered pair $(v_i, v_j) \in \mathcal{V} \times \mathcal{V}$ and each parameter pair $(p_o, p_i) \in \mathcal{O}(v_i) \times \mathcal{I}(v_j)$ whose cosine similarity exceeds a preset threshold, we add the edge $(v_i \xrightarrow{p_i} v_j)$. Because matching runs over the global toolset, flow edges may connect tools of different environments, which is what allows a sampled plan to span multiple servers. Details are provided in Appendix D.

Step 2: Logical Dependency Augmentation. Semantic matching cannot connect tools whose relation is not carried by a shared parameter; parameter-less tools, in particular, would remain isolated. For each environment $e \in \mathcal{E}$, we therefore prompt an LLM with the tools in $\mathcal{V}_e$ and the current adjacency map, and it proposes workflow edges $(v_i \to v_j) \in E_{\text{wf}}$ justified by data-flow feasibility or a natural usage order. A workflow edge is never taken as evidence that a specific input is produced.

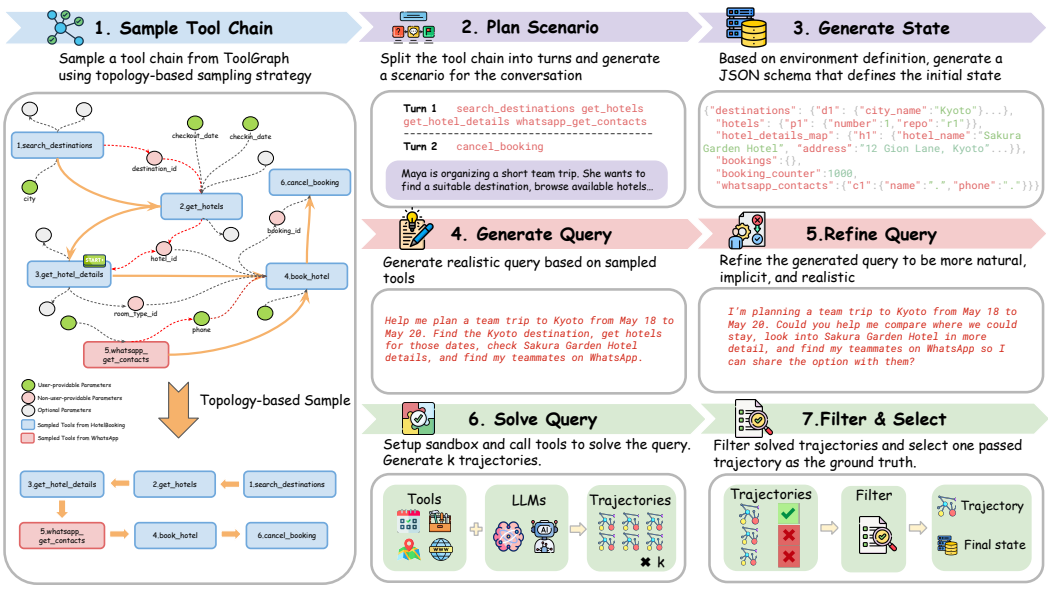


Figure 3: The overall framework of *QueryGen*: Part 1 constructs dependency-complete, non-linear plans through topology-aware sampling; Parts 2–5 ground scenarios and inject implicitness into requests; Parts 6–7 verify executions and select an execution-grounded reference trajectory.

3.3.2 Topology-Aware Sampling

Given G, we sample an ordered tool plan $\tau = [v_1, \ldots, v_n]$ that serves as the backbone of a query. A request built on τ must be solvable: every required input of every tool in τ must be obtainable from the user or from the output of an earlier tool in τ. Random walks do not check this, and the resulting unsatisfiable requests are the main source of the all-failure tasks in Figure 1a. Realistic workflows are also rarely linear. The sampler therefore admits a tool into τ only after its inputs are resolved, and then expands the plan across its neighbors (Figure 3, Part 1) to carry non-linearity.

Identifying Internal and External Parameters. We classifies each input parameter as *external* or *internal*. External parameters (e.g., `city`, `name`) describe information a human user can naturally supply; internal parameters (e.g., `hotel_id` for `book_hotel`) denote system state that a user is unlikely to know and that must come from a preceding tool call (e.g., `get_hotel_list`). The same classification later bounds what the simulated user may reveal (Appendix C.2).

Resolving Dependencies. When considering a tool v_i in the tool plan $\tau = [v_1, ..., v_n]$, we will chech whether the dependencies of all input parameters $p \in \mathcal{I}(v_i)$ are resolved first. If p: (1) is *optional*, with a default value or omittable; (2) is *external*, so the user can supply it; or (3) is *internal* and some earlier tool v_j in τ has a flow edge $(v_j \xrightarrow{p} v_i)$, then we identify it as resolved. For every unresolved parameter, we samples the producer v_j with $(v_j \xrightarrow{p} v_i) \in E_{\text{flow}}$ and resolves the inputs of v_j recursively, up to a maximum depth $D_{\max}$. This recursive sampling ensure all dependencies in the tool plan $\tau = [v_1, ..., v_n]$ can be resolved. To diversify plans, we may additionally insert a producer for an already satisfied parameter with a low probability ρ. Full algorithms in 1.

Expanding Neighbors. Once v_i is admitted, we draws 1 to b of its unvisited successors uniformly at random along outgoing edges in $E_{\text{flow}} \cup E_{\text{wf}}$ and queues them for the same treatment, so plans branch non-linearly rather than sequentially.

3.4 Trajectory Synthesis with Implicitness

Overview. Given a dependency-complete plan τ, *QueryGen* converts it into a state-grounded, multi-turn trajectory while controlling how much of the latent execution plan appears in the user request. Directly verbalizing every tool and argument would produce an over-specified instruction list and favor all-success outcomes. We instead preserve the intended outcome while leaving inferable actions

and identifiers implicit. The resulting task must remain *solvable*, first by construction (Section 3.3.2) and then by execution, while becoming *realistic yet difficult* through reduced surface cues.

State-Grounded Planning. From τ, we construct a user profile and scenario, then instantiate the initial state s_0 of every involved environment using the schemas of Section 3.2. The scenario and records are grounded in web search and sample values retrieved during environment construction. Shared entities (e.g., the user, dates, and locations) are drafted jointly, and database tables are populated in dependency order to maintain cross-environment consistency. Finally, we stochastically partition τ into dialogue turns of 1–5 tools each.

Implicitness Injection. For each turn, *QueryGen* first maps the tools to fine-grained subgoals and intents, then articulates an explicit draft conditioned on the current database state and dialogue history. This draft anchors the requested outcome before four operations reduce its surface cues: (1) *implicit reference* replaces explicit identifiers with contextual references and omits deducible parameters; (2) *action compression* removes logically inferable intermediate steps; (3) *ambiguity introduction* adds reasonable referential ambiguity; and (4) *goal expansion* adds plausible, thematically related objectives. These transformations change how directly the solution is stated without severing its grounding in τ and s_0, requiring the agent to recover references and intermediate actions from context, state, and tool dependencies.

Agentic Verification. Because implicitness reduces explicit cues, we recheck solvability by letting an agent interact with the sandboxed environments and a simulated user under the protocol of Section 3.1. For each query, we run k independent rollouts to cover alternative solutions, and a rubric-based LLM judge inspects each trajectory ξ and terminal state s_T. A rollout passes if it (i) addresses every explicit requirement; (ii) reaches a fulfilling terminal state with no spurious writes; (iii) asks the user only for information that no tool can retrieve; and (iv) ends with a clear completion summary. We drop a task if all rollouts fail and retain the best successful rollout as its execution-grounded reference trajectory for SFT and RL. Finally, filtering removes redundant calls and user turns, while masking marks arguments whose values do not affect correctness.

3.5 Model Training

With the synthesized trajectories, we post-train models with SFT followed by RL. For RL, tool-use correctness is hard to verify because valid executions are non-unique: independent read-only calls may be issued in any order, arguments such as `limit` may vary, and a read-only task leaves the database unchanged, so neither a reference trajectory nor a terminal state suffices alone. We therefore combine a *trajectory reward* $R_{\text{traj}} \in \{0, 1\}$, which is 1 iff every call of the reference trajectory is matched by a distinct rollout call under set-based matching that ignores arguments masked in Section 3.4; a *state reward* $R_{\text{state}} \in [0, 1]$, the per-server average of exact terminal-state matches to the reference state; and a capped *length penalty* P_{length} on redundant and malformed calls:

$$R = \max\big(0,\ \alpha R_{\text{traj}} + (1-\alpha) R_{\text{state}} - \gamma P_{\text{length}}\big),$$

The ablation study in Section 4.4 demonstrates the necessity for both trajectory and state reward. More details are provided in Appendix C.3.

4 Experiments

4.1 Setup

Data Statistics. We construct 85 MCP environments spanning 7 domains (commerce, finance, travel, office, lifestyle, research, and utilities) and synthesize 1,622 multi-turn SFT trajectories and 953 single-request RL trajectories. The SFT conversations average 4.82 turns, with 3.29 steps per turn; Figure 6 reports their distributions. We count each generated episode as one trajectory: a multi-turn conversation for SFT and a single-request episode for RL. For training, SFT converts each step into one supervision example, while each RL request is paired with its reference tool calls and terminal state (Appendix D).

Baselines and Benchmarks. We adopt Qwen3-(1.7B, 4B, 8B) (Yang et al., 2025a) as training backbones and compare with the released checkpoints of AWM (Wang et al., 2026) and EnvScaler (Song et al., 2026), two concurrent works on tool-use trajectory synthesis. We evaluate on BFCL-v3 (Patil

Table 1: Results on four tool-use benchmarks, including the retained MCP-Atlas subset. **Cell** / Cell : best / second-best metric at each scale. **Cell** / Cell : fewest / second-fewest reported environments or training-data samples among trained checkpoints. Overall calculation is described in Appendix B.

Model	Data Scale		BFCL		MCP-Atlas		τ^2-Bench				VitaBench				Overall
	Env.	*Data*	*Single Turn*	*Multi Turn*	*Pass Rate*	*Mean Cov.*	*Airline*	*Retail*	*Telecom*	**Avg.**	*Delivery*	*Store*	*OTA*	**Avg.**	*Avg.*
Qwen3-1.7B															
Base	–	–	**79.48**	16.75	1.03	6.25	14.00	7.02	22.81	14.61	4.00	0.00	0.00	1.33	16.27
EnvScaler	191	11,572	60.41	**30.13**	2.75	9.40	12.00	18.42	10.53	13.65	9.00	3.00	1.09	4.36	16.51
Ours (SFT)	**85**	**1,622**	78.30	23.25	1.72	**10.05**	16.00	20.18	10.53	**15.57**	13.00	6.00	0.00	6.33	18.60
Ours (SFT+RL)	**85**	2,575	78.44	28.38	**3.09**	9.64	12.00	16.67	16.67	15.11	11.00	11.00	0.00	**7.33**	**19.74**
Qwen3-4B															
Base	–	–	85.15	33.50	4.12	12.86	24.00	38.60	13.16	25.25	9.00	12.00	2.02	7.67	24.09
AWM	526	3,315	**85.97**	40.75	4.47	12.33	18.00	31.58	17.54	22.37	22.00	13.00	0.00	11.67	25.47
EnvScaler	191	11,572	83.64	45.00	**9.97**	**22.27**	36.00	41.23	10.53	29.25	23.00	15.00	6.06	14.69	29.56
Ours (SFT)	**85**	**1,622**	85.10	44.25	7.90	19.66	24.00	47.37	4.39	25.25	19.00	12.00	3.00	11.33	27.29
Ours (SFT+RL)	**85**	2,575	85.46	**48.50**	**9.97**	21.89	36.00	42.11	12.28	**30.13**	21.00	21.00	6.00	**16.00**	**30.77**
Qwen3-8B															
Base	–	–	84.31	41.25	5.15	14.86	32.00	42.98	21.93	32.30	24.00	15.00	11.11	16.70	29.23
AWM	526	3,315	84.80	42.25	6.19	16.60	30.00	29.82	25.44	28.42	20.00	15.00	14.43	16.48	28.65
EnvScaler	191	11,572	84.74	**51.88**	9.62	22.63	38.00	49.12	15.79	**34.30**	25.00	19.00	12.00	**18.67**	32.72
Ours (SFT)	**85**	**1,622**	84.83	46.50	8.25	22.86	42.00	43.86	12.28	32.71	23.00	20.00	7.00	16.67	30.82
Ours (SFT+RL)	**85**	2,575	**86.02**	49.00	**13.75**	**25.98**	44.00	43.86	13.16	33.67	24.00	22.00	10.00	**18.67**	**33.40**

et al., 2025), τ^2-Bench (Barres et al., 2025), VitaBench (He et al., 2026b), and MCP-Atlas (Bandi et al., 2026), which together cover structured function calling, stateful multi-turn interaction, user–agent coordination, long-horizon daily-service tasks, and cross-server MCP workflows. Appendix B describes each benchmarks in details.

Implementation Details. Training has two stages: *Stage 1* performs SFT on the conversation-level trajectories, including user turns; *Stage 2* performs RL on tool-call-only episodes without a simulated user, initialized from the checkpoint after SFT. We use LlamaFactory (Zheng et al., 2024b) for SFT and VeRL (Sheng et al., 2025) with GRPO (Shao et al., 2024) for RL; details are provided in Appendix D.

4.2 Main Results

Table 1 reports results on the four benchmarks at three model scales.

SFT Cold Start Delivers the Largest Relative Gains. SFT on our trajectories alone raises the Overall score at every scale, with gains concentrated on multi-turn and cross-server tasks. BFCL multi-turn accuracy win the largest gain from 33.50 to 44.25 at 4B; MCP-Atlas pass rate rises at every scale (e.g., 4.12 to 7.90 at 4B); VitaBench improves from 1.33 to 6.33 at 1.7B and from 7.67 to 11.33 at 4B; and τ^2-Bench Retail improves at the two smaller scales (4B: 38.60 to 47.37). Overall, EnvFactory (SFT) consistently improves average performance across all model scales, demonstrating that our synthesized trajectories provide an effective cold-start signal for scalable tool-use learning.

RL after SFT Further Unlocks Tool-Use Capability. Further RL raises the overall score above the reported three-epoch SFT model at all three scales: from 18.60 to 19.74 at 1.7B, from 27.29 to 30.77 at 4B, and from 30.82 to 33.40 at 8B. The largest gains appear on the interactive and cross-server benchmarks: VitaBench at 4B (11.33 to 16.00), MCP-Atlas at 8B (pass rate 8.25 to 13.75), and BFCL multi-turn at 4B and 8B (44.25 to 48.50; 46.50 to 49.00). The gains hold on both conversational (τ^2-Bench, VitaBench) and non-conversational (BFCL, MCP-Atlas) benchmarks, so SFT supplies the tool-use behaviors and RL further improves execution on stateful, multi-step tasks. We also include evaluation on out-of-domain math results in Appendix A.2.

Comparison with Released Baselines. Across the 21 headline cells of Table 1, EnvFactory (SFT+RL) leads the base model and released checkpoints in 13 and ties in 2; it achieves the best

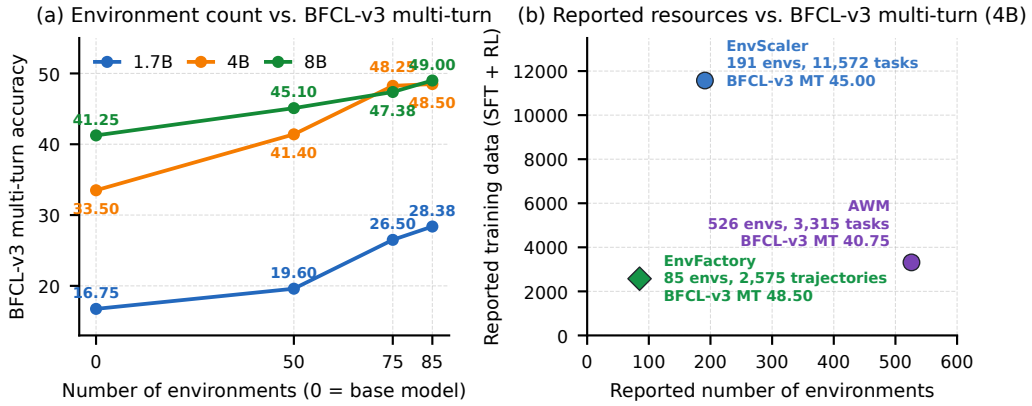


Figure 4: (a) BFCL-v3 multi-turn accuracy by environment count and model scale. (b) Released 4B checkpoints by reported environment and training-data counts; labels show BFCL-v3 multi-turn scores.

Table 2: Sampling strategies with 7,000 SFT trajectories each; other stages fixed.

Model	BFCL Multi-turn	τ^2-Bench	VitaBench
Random Walk-1.7B	16.12	**15.49**	2.00
Topology-Aware-1.7B	**17.00**	13.27	**2.67**
Random Walk-4B	37.00	25.33	11.35
Topology-Aware-4B	**39.50**	**28.13**	**12.04**
Random Walk-8B	42.88	35.23	18.66
Topology-Aware-8B	**43.88**	**37.39**	**19.00**

Table 3: Effect of implicitness injection on BFCL-v3 multi-turn.

Setting	Base	Miss Func	Miss Param	Long Context	Overall
Original-1.7B	30.0	21.5	19.5	14.0	21.25
Injected-1.7B	**30.5**	**22.5**	**21.0**	**14.5**	**22.12**
Original-4B	**52.0**	47.0	30.5	34.0	40.88
Injected-4B	49.5	**47.5**	**32.0**	**36.0**	**41.25**
Original-8B	51.5	47.0	38.5	35.0	43.00
Injected-8B	**55.0**	47.0	**39.0**	35.0	**44.00**

Overall score at every scale. Comparing with EnvScaler and AWM, ENVFACTORY uses fewer environments (85 v.s. 191 v.s. 526), demonstrating better effectiveness and efficiency. These results demonstrate that ENVFACTORY generalizes effectively across both conversational interaction and compositional tool-execution settings.

Capability Retention. We further evaluate ENVFACTORY on math benchmarks to assess out-of-domain capability retention after tool-use post-training. Based on Table 5, SFT+RL and SFT alone maintain math performance comparable to the base models. Details are provided in Appendix A.

4.3 ABLATIONS ON DATA SYNTHESIS

Effect of Environment Scaling. Figure 4(a) studies how the number of executable environments affects BFCL-v3 multi-turn performance. We construct two additional training subsets with 50 and 75 randomly sampled environments and apply the same SFT+RL procedure to each. Performance rises with the environment count at all three model scales, and the gain from 50 to 75 environments exceeds the gain from 75 to 85, suggesting that early expansion adds new tool schemas and interaction patterns faster than later additions. Each point uses a single sampled subset, and trajectory volume grows with the environment count, so the curve reflects the combined effect of environment diversity and data volume. Figure 4(b) places the 4B checkpoints of ENVFACTORY, AWM, and EnvScaler by their reported environment and training-data counts: ENVFACTORY reaches the highest BFCL multi-turn score with the fewest environments and the smallest reported data count.

Effect of Topology-Aware Sampling. We synthesize 7,000 SFT trajectories with naive random-walk traversal and 7,000 with topology-aware sampling, keeping the rest of the pipeline fixed, and fine-tune each model scale on each set. Table 2 shows that topology-aware sampling improves all three benchmarks at 4B and 8B and two of three benchmarks at 1.7B, yielding gains in 8 of 9 comparisons overall. Because topology-aware sampling jointly determines the tool chain, query,

Table 4: Direct RL from the base model, without SFT. Bold: the better of the base model and direct RL at each scale.

Model	BFCL Single-turn	BFCL Multi-turn	τ^2 Bench	VitaBench
Qwen3-1.7B	79.48	16.75	14.61	1.33
Ours-1.7B (direct RL)	**79.53**	**18.33**	**18.28**	**1.67**
Qwen3-4B	85.15	33.50	**25.25**	7.67
Ours-4B (direct RL)	**85.26**	**41.38**	24.83	**12.74**
Qwen3-8B	84.31	41.25	**32.30**	16.70
Ours-8B (direct RL)	**84.42**	**44.35**	29.08	**17.00**



Figure 5: BFCL-v3 multi-turn accuracy during 4B direct RL by trajectory-reward weight α.

and initial state, this experiment evaluates the complete training distribution it induces. Figure 1a gives the mechanism-level view, where dependency resolution lowers the share of all-failure tasks from 15% to 9%.

Effect of Implicitness Injection. We synthesize 250 SFT trajectories with and without implicitness injection and evaluate on the BFCL multi-turn categories (Patil et al., 2025) (`Base`, `Miss-Func`, `Miss-Param`, and `Long-Context`). Table 3 shows that implicitness injection improves the category average at all three scales and wins or ties 11 of the 12 category-scale comparisons. `Miss-Param`, which requires the model to notice that information is missing and ask for it, improves at every scale. This controlled comparison evaluates downstream utility, while the human evaluation in Appendix A.1 complements it with annotators preferring the injected queries for naturalness in 71% of paired comparisons.

4.4 Ablations on Training

Effect of Direct RL. We examine whether EnvFactory trajectories can support RL without an SFT cold start. Table 4 shows that direct RL improves BFCL multi-turn and VitaBench at every scale (e.g., BFCL multi-turn at 4B from 33.50 to 41.38), but the gains are smaller than those of RL after SFT (Table 1): at 4B, direct RL reaches 41.38 on BFCL multi-turn against 48.50 for SFT+RL, and at 4B and 8B it lowers τ^2-Bench. At 1.7B, direct RL exceeds SFT+RL on BFCL single-turn (79.53 vs. 78.44) and τ^2-Bench (18.28 vs. 15.11), so the cold start matters most for multi-turn execution at the larger scales. Inspection of early rollouts points to the mechanism: malformed tool-call syntax and calls to nonexistent tools appear in roughly 30% of early direct-RL rollouts despite the base models' tool-use post-training, and SFT initialization roughly halves this rate.

Effect of Reward Weighting. We sweep the trajectory-reward weight $\alpha \in \{0, 0.3, 0.5, 0.7, 1.0\}$ with direct RL at the 4B scale, fixing $\gamma = 1$, and track BFCL-v3 multi-turn accuracy during training (Figure 5). State-only ($\alpha = 0$) and trajectory-only ($\alpha = 1$) reach 39.38 and 38.62, respectively, whereas combining the rewards reaches 40.87 at $\alpha = 0.3$ and 41.38 at $\alpha = 0.5$. Removing either component therefore reduces performance, showing that trajectory matching and state equivalence provide complementary signals and that both are needed for the strongest training outcome.

5 Conclusion

We presented EnvFactory, which builds stateful tool environments from retrieved documentation and synthesizes multi-turn trajectories. Dependency-aware sampling and implicitness injection raise the share of mixed-outcome tasks from 24% to 56%. Training on 85 environments and 2,575 trajectories consistently improves aggregate performance across four tool-use benchmarks and three Qwen3 model scales.

AI Use Statement

Generative AI was used in two roles. As part of the research pipeline, it assisted in constructing and validating executable environments and in generating, refining, and filtering synthetic training trajectories; the resulting data were then used to train the models evaluated in this work. Sepa-

rately, generative AI assistants were used to improve the manuscript's language, organization, and formatting. Generative AI was not used to originate the research ideas or formulate hypotheses; all research ideas, methodological decisions, and experimental designs were developed by the authors. All AI-assisted manuscript edits were reviewed and approved by the authors, who verified the reported results and take full responsibility for the paper's content and all AI-assisted artifacts.

Ethics Statement

This work studies executable synthetic environments rather than deployment against live services. Generated code and write-capable tools may create security, privacy, authorization, and destructive-action risks if deployed without appropriate safeguards. Our experiments operate in isolated synthetic environments.

Reproducibility Statement

Section 3 specifies the pipeline; Appendices C.1 and C.3 give the sampling algorithms and reward computation; Appendix D gives the synthesis models and training setup, with key hyperparameters in Table 8; Appendix B describes the benchmarks and metrics; Appendix A.1 describes the human evaluation protocol; and Appendix G provides the included environment-construction, graph-augmentation, and query-generation prompts. We will release the code, the 85 environments, the synthesized trajectories, and the trained checkpoints.

References

Anthropic. Introducing the model context protocol. https://www.anthropic.com/news/model-context-protocol, November 2024. Accessed: 2026-05-05.

Chaithanya Bandi, Razvan-Gabriel Dumitru, Ben Hertzberg, Divyansh Agarwal, Geobio Boo, Tejas Polakam, Sami Hassaan, Jeff Da, HiJae Kim, Vipul Gupta, Manasi Sharma, Andrew Park, Martin Dimakis, Ernesto Gabriel Hernandez Montoya, Dan Rambado, Ivan Salazar, Rafael Cruz, MohammadHossein Rezaei, Chetan Rane, Ben Levin, Daniel Yue Zhang, Brad Kenstler, and Bing Liu. Mcp-atlas: A large-scale benchmark for tool-use competency with real mcp servers, 2026. URL https://arxiv.org/abs/2602.00933.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. τ^2-bench: Evaluating conversational agents in a dual-control environment, 2025. URL https://arxiv.org/abs/2506.07982.

Shihao Cai, Runnan Fang, Jialong Wu, Baixuan Li, Xinyu Wang, Yong Jiang, Liangcai Su, Liwen Zhang, Wenbiao Yin, Zhen Zhang, Fuli Feng, Pengjun Xie, and Xiaobin Wang. Autoforge: Automated environment synthesis for agentic reinforcement learning, 2025. URL https://arxiv.org/abs/2512.22857.

Jianlv Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. M3-embedding: Multi-linguality, multi-functionality, multi-granularity text embeddings through self-knowledge distillation. In *Findings of the Association for Computational Linguistics: ACL 2024*, 2024. doi: 10.18653/v1/2024.findings-acl.137. URL https://aclanthology.org/2024.findings-acl.137/.

Wenjie Chen, Wenbin Li, Di Yao, Xuying Meng, Chang Gong, and Jingping Bi. Gtool: Graph enhanced tool planning with large language model, 2025. URL https://arxiv.org/abs/2508.12725.

Zhaorun Chen, Zhuokai Zhao, Kai Zhang, Bo Liu, Qi Qi, Yifan Wu, Tarun Kalluri, Sara Cao, Yuanhao Xiong, Haibo Tong, Huaxiu Yao, Hengduo Li, Jiacheng Zhu, Xian Li, Dawn Song, Bo Li, Jason Weston, and Dat Huynh. Scaling agent learning via experience synthesis, 2026. URL https://arxiv.org/abs/2511.03773. International Conference on Learning Representations (ICLR).

DeepSeek-AI et al. DeepSeek-V3.2: Pushing the frontier of open large language models, 2025. URL https://arxiv.org/abs/2512.02556.

DeepSeek-AI et al. DeepSeek-V4: Towards highly efficient million-token context intelligence, 2026. URL https://arxiv.org/abs/2606.19348.

Runnan Fang, Shihao Cai, Baixuan Li, Jialong Wu, Guangyu Li, Wenbiao Yin, Xinyu Wang, Xiaobin Wang, Liangcai Su, Zhen Zhang, Shibin Wu, Zhengwei Tao, Yong Jiang, Pengjun Xie, Fei Huang, and Jingren Zhou. Towards general agentic intelligence via environment scaling, 2025. URL https://arxiv.org/abs/2509.13311.

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, 2026. URL https://arxiv.org/abs/2504.11536. International Conference on Learning Representations (ICLR).

Bingguang Hao, Zengzhuang Xu, Maolin Wang, Yuntao Wen, Yicheng Chen, Cunyin Peng, Long Chen, Dong Wang, Xiangyu Zhao, Jinjie Gu, Chenyi Zhuang, and Ji Zhang. Reasoning through exploration: A reinforcement learning framework for robust function calling, 2025. URL https://arxiv.org/abs/2508.05118.

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

Pengfei He, Zhenwei Dai, Bing He, Hui Liu, Xianfeng Tang, Hanqing Lu, Juanhui Li, Jiayuan Ding, Subhabrata Mukherjee, Suhang Wang, Yue Xing, Jiliang Tang, and Benoit Dumoulin. TRAJECT-Bench: A trajectory-aware benchmark for evaluating agentic tool use, 2026a. URL https://arxiv.org/abs/2510.04550. International Conference on Learning Representations (ICLR).

Wei He, Yueqing Sun, Hongyan Hao, Xueyuan Hao, Zhikang Xia, Qi Gu, Chengcheng Han, Dengchang Zhao, Hui Su, Kefeng Zhang, Man Gao, Xi Su, Xiaodong Cai, Xunliang Cai, Yu Yang, and Yunke Zhao. Vitabench: Benchmarking llm agents with versatile interactive tasks in real-world applications, 2026b. URL https://arxiv.org/abs/2509.26490. International Conference on Learning Representations (ICLR).

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, 2025. URL https://arxiv.org/abs/2503.09516. Conference on Language Modeling (COLM).

Adam Tauman Kalai, Ofir Nachum, Santosh S. Vempala, and Edwin Zhang. Why language models hallucinate, 2025. URL https://arxiv.org/abs/2509.04664.

Kimi Team et al. Kimi K2: Open agentic intelligence, 2025. URL https://arxiv.org/abs/2507.20534.

Sijia Li, Yuchen Huang, Zifan Liu, Zijian Li, Jingjing Fu, Lei Song, Jiang Bian, Jun Zhang, and Rui Wang. Experience-evolving multi-turn tool-use agent with hybrid episodic-procedural memory, 2026a. URL https://arxiv.org/abs/2512.07287. International Conference on Machine Learning (ICML).

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?, 2026b. URL https://aclanthology.org/2026.acl-long.366/. Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (ACL 2026).

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, 2025. URL https://arxiv.org/abs/2511.01824.

Xiao Liang, Zhongzhi Li, Yeyun Gong, Yelong Shen, Ying Nian Wu, Zhijiang Guo, and Weizhu Chen. Beyond pass@ 1: Self-play with variational problem synthesis sustains rlvr. *arXiv preprint arXiv:2508.14029*, 2025.

Junyu Luo, Weizhi Zhang, Ye Yuan, Yusheng Zhao, Junwei Yang, Yiyang Gu, Bohan Wu, Binqi Chen, Ziyue Qiao, Qingqing Long, Rongcheng Tu, Xiao Luo, Wei Ju, Zhiping Xiao, Yifan Wang, Meng Xiao, Chenwu Liu, Jingyang Yuan, Shichang Zhang, Yiqiao Jin, Fan Zhang, Xian Wu, Hanqing Zhao, Dacheng Tao, Philip S. Yu, and Ming Zhang. Large language model agent: A survey on methodology, applications and challenges, 2025. URL https://arxiv.org/abs/2503.21460.

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

Shishir G. Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. The berkeley function calling leaderboard (BFCL): from tool use to agentic evaluation of large language models. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*, Proceedings of Machine Learning Research. PMLR / OpenReview.net, 2025. URL https://proceedings.mlr.press/v267/patil25a.html.

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 *The Twelfth International Conference on Learning Representations*, 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), January 2025. ISSN 2095-2236. doi: 10.1007/s11704-024-40678-2. URL http://dx.doi.org/10.1007/s11704-024-40678-2.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in neural information processing systems*, 36:53728–53741, 2023.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. URL https://arxiv.org/abs/2402.03300.

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*, 2025. doi: 10.1145/3689031.3696075. URL https://doi.org/10.1145/3689031.3696075.

Xiaoshuai Song, Haofei Chang, Guanting Dong, Yutao Zhu, Ji-Rong Wen, and Zhicheng Dou. Envscaler: Scaling tool-interactive environments for llm agent via programmatic synthesis, 2026. URL https://arxiv.org/abs/2601.05808.

Michael Sullivan, Mareike Hartmann, and Alexander Koller. Procedural environment generation for tool-use agents. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, 2025. doi: 10.18653/v1/2025.emnlp-main.936. URL https://aclanthology.org/2025.emnlp-main.936/.

Dunwei Tu, Hongyan Hao, Hansi Yang, Yihao Chen, Yi-Kai Zhang, Zhikang Xia, Yu Yang, Yueqing Sun, Xingchen Liu, Furao Shen, Qi Gu, Hui Su, and Xunliang Cai. Scaleenv: Scaling environment synthesis from scratch for generalist interactive tool-use agent training, 2026. URL https://arxiv.org/abs/2602.06820.

Ruoyao Wang, Graham Todd, Ziang Xiao, Xingdi Yuan, Marc-Alexandre Côté, Peter Clark, and Peter Jansen. Can language models serve as text-based world simulators? In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 1–17, 2024. doi: 10.18653/v1/2024.acl-short.1. URL https://aclanthology.org/2024.acl-short.1/.

Zezhong Wang, Xingshan Zeng, Weiwen Liu, Liangyou Li, Yasheng Wang, Lifeng Shang, Xin Jiang, Qun Liu, and Kam-Fai Wong. Toolflow: Boosting llm tool-calling through natural and coherent dialogue synthesis. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics*, 2025. doi: 10.18653/v1/2025.naacl-long.214. URL https://aclanthology.org/2025.naacl-long.214/.

Zhaoyang Wang, Canwen Xu, Boyi Liu, Yite Wang, Siwei Han, Zhewei Yao, Huaxiu Yao, and Yuxiong He. Agent world model: Infinity synthetic environments for agentic reinforcement learning, 2026. URL https://arxiv.org/abs/2602.10090. International Conference on Machine Learning (ICML).

Jiaqi Wu, Qinlao Zhao, Zefeng Chen, Kai Qin, Yifei Zhao, Xueqian Wang, and Yuhang Yao. Gap: Graph-based agent planning with parallel tool use and reinforcement learning, 2025. URL https://arxiv.org/abs/2510.25320.

Zhangchen Xu, Adriana Meza Soria, Shawn Tan, Anurag Roy, Ashish Sunil Agrawal, Radha Poovendran, and Rameswar Panda. Toucan: Synthesizing 1.5m tool-agentic data from real-world mcp environments, 2025. URL https://arxiv.org/abs/2510.01179.

An Yang et al. Qwen3 technical report, 2025a. URL https://arxiv.org/abs/2505.09388.

Chen Yang, Ran Le, Yun Xing, Zhenwei An, Zongchao Chen, Wayne Xin Zhao, Yang Song, and Tao Zhang. Toolmind technical report: A large-scale, reasoning-enhanced tool-use dataset, 2025b. URL https://arxiv.org/abs/2511.15718.

Zhicheng Yang, Zhijiang Guo, Yinya Huang, Xiaodan Liang, Yiwei Wang, and Jing Tang. Treerpo: Tree relative policy optimization. *arXiv preprint arXiv:2506.05183*, 2025c.

Zhicheng Yang, Zhijiang Guo, Yinya Huang, Yongxin Wang, Dongchun Xie, Hanhui Li, Yiwei Wang, Xiaodan Liang, and Jing Tang. Depth-breadth synergy in rlvr: Unlocking llm reasoning gains with adaptive exploration. *arXiv preprint arXiv:2508.13755*, 2025d.

Junjie Ye, Changhao Jiang, Zhengyin Du, Yufei Xu, Xuesong Yao, Zhiheng Xi, Xiaoran Fan, Qi Zhang, Tao Gui, Xuanjing Huang, and Jiecao Chen. Feedback-driven tool-use improvements in large language models via automated build environments, 2026. URL https://aclanthology.org/2026.findings-acl.109/. Findings of the Association for Computational Linguistics: ACL 2026.

Fan Yin, Zifeng Wang, I-Hung Hsu, Jun Yan, Ke Jiang, Yanfei Chen, Jindong Gu, Long T. Le, Kai-Wei Chang, Chen-Yu Lee, Hamid Palangi, and Tomas Pfister. Magnet: Multi-turn tool-use data synthesis and distillation via graph translation, 2025. URL https://arxiv.org/abs/2503.07826.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, et al. Dapo: An open-source llm reinforcement learning system at scale, 2025. URL https://arxiv.org/abs/2503.14476.

Yang Yue, Zhiqi Chen, Rui Lu, Andrew Zhao, Zhaokai Wang, Yang Yue, Shiji Song, and Gao Huang. Does reinforcement learning really incentivize reasoning capacity in llms beyond the base model? *Advances in Neural Information Processing Systems*, 2025.

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

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. LLaMA-Factory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 3: system demonstrations)*, pp. 400–410, 2024b. URL https://aclanthology.org/2024.acl-demos.38/.

Xinyu Zhu, Mengzhou Xia, Zhepei Wei, Wei-Lin Chen, Danqi Chen, and Yu Meng. The surprising effectiveness of negative reinforcement in llm reasoning. *Advances in Neural Information Processing Systems*, 2025.

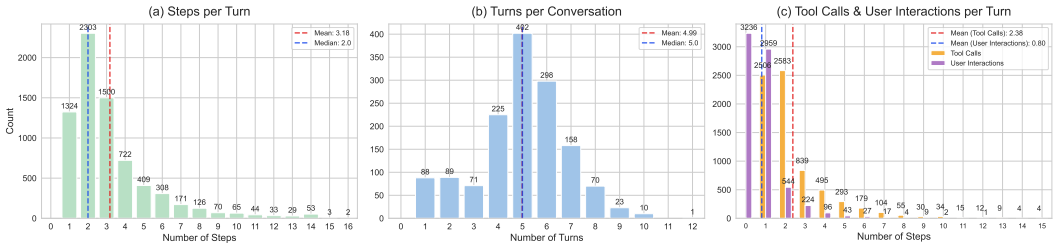


Figure 6: **Data statistics for retained SFT conversations.** The plotted subset averages 4.99 turns per conversation and 3.18 steps per turn, closely tracking the full-dataset averages of 4.82 and 3.29 reported in Section 4.1. (a) Steps per turn. (b) Turns per conversation. (c) Tool calls and user interactions per turn.

A Additional Experiments

A.1 Human Evaluation

To complement automated validation and model-based trajectory selection, we conduct a two-stage human evaluation of the quality and realism of the generated environments and queries.

Environment Evaluation. Eight annotators with graduate training in computer science participated. Each annotator was randomly and independently assigned three constructed environments and evaluated them at the environment, tool, and parameter levels. The rubric assessed functional correctness, specification clarity, tool-set coherence, and whether the APIs collectively support the modeled environment.

Query Evaluation. We also randomly sampled 20 queries for paired evaluation, presenting each implicitness-injected query alongside its unrefined counterpart and execution-grounded reference trajectory. Using criteria aligned with QueryGen's request-generation and reference-trajectory selection stages, annotators assessed request solvability, naturalness, and execution efficiency. Across paired comparisons, annotators preferred the implicitness-injected queries for naturalness in 71% of cases.

A.2 Out-of-Domain Evaluation on Math

Telecom is out of distribution relative to our tool-centric training setup: it is a dual-control domain in which the agent must guide the simulated user to act on a shared environment, rather than execute every action through its own tools. To assess capability retention beyond tool use, we evaluate mathematical reasoning on AIME24, AIME25, and MATH-500. In Table 5, SFT+RL exceeds the base average by 1.10 points at 1.7B and 1.42 at 4B; at 8B it remains 1.50 points below base while recovering part of the 2.29-point drop after SFT. Across scales, SFT+RL therefore preserves average math performance within 1.5 points of the base models, with gains at 1.7B and 4B.

Table 5: Out-of-domain math performance with a 32k token budget.

Size	Variant	AIME24	AIME25	MATH-500	Avg.	Δ vs. Base
1.7B	Qwen3-1.7B (Base)	47.50	40.00	90.30	59.27	–
	Ours (SFT)	51.67	34.17	91.40	59.08	−0.19
	Ours (SFT+RL)	52.50	36.67	91.95	60.37	+1.10
4B	Qwen3-4B (Base)	73.02	65.42	95.40	77.95	–
	Ours (SFT)	74.27	66.77	95.50	78.85	+0.90
	Ours (SFT+RL)	75.21	67.71	95.20	79.37	+1.42
8B	Qwen3-8B (Base)	75.83	70.00	95.80	80.54	–
	Ours (SFT)	73.33	65.83	95.60	78.25	−2.29
	Ours (SFT+RL)	75.00	66.67	95.45	79.04	−1.50

B Evaluation Details

B.1 Setup

During inference, we use SGLang (Zheng et al., 2024a) with tensor parallelism set to 2. EnvFactory (SFT and SFT+RL), AWM, and EnvScaler are decoded in thinking mode at temperature 0.7. We use DeepSeek-V4-Flash-Preview (DeepSeek-AI et al., 2026) in non-thinking mode at temperature 0 to simulate users and evaluate responses in τ^2-Bench and VitaBench, and to judge answers in MCP-Atlas.

B.2 Benchmarks

BFCL-v3. BFCL evaluates structured function calling across complementary single-turn and multi-turn settings (Patil et al., 2025). The single-turn track tests function selection and argument generation, including serial, parallel, and relevance-detection cases. The multi-turn track executes tools in stateful tasks that require planning, maintaining interaction state, handling missing parameters or unavailable functions, and operating under long context. We report the benchmark's aggregate single-turn and multi-turn accuracies.

τ^2-Bench. τ^2-Bench evaluates policy-grounded, multi-turn coordination between an agent, a simulated user, and stateful tools (Barres et al., 2025). Our evaluation covers Airline, Retail, and Telecom. Airline and Retail test interactive service tasks in which the agent must gather information and act under domain policies; Telecom additionally uses a dual-control setting in which both the agent and user can act on the shared environment. We report task success for each domain and their unweighted average.

VitaBench. VitaBench evaluates long-horizon interactive tasks drawn from food delivery, in-store consumption, and online travel applications (He et al., 2026b). Its scenarios combine multiple user requests and test temporal and spatial reasoning, tool selection, clarification of ambiguous instructions, and adaptation to changing intent. We report task success for the Delivery, Store, and OTA domains and their unweighted average.

MCP-Atlas. MCP-Atlas evaluates tool-use competency on real MCP servers through multi-step, cross-server workflows (Bandi et al., 2026). Prompts do not identify the required servers, tools, or parameters, and answer quality is scored against atomic claims grounded in tool outputs using DeepSeek-V4-Flash-Preview (DeepSeek-AI et al., 2026) in non-thinking mode. We report Mean Coverage, the average fraction of reference claims satisfied, and Pass Rate, the fraction of tasks satisfying the benchmark's claim-coverage pass criterion.

B.3 Overall Score

The Overall column in Table 1 is a descriptive aggregate that gives equal weight to the four benchmark families. We first average BFCL single-turn and multi-turn accuracy, and then average that BFCL value with MCP-Atlas Pass Rate, the τ^2-Bench domain average, and the VitaBench domain average. MCP-Atlas Mean Coverage is reported as a diagnostic and is not included. Overall is provided only for compact comparison across the displayed metrics; it is not an independent evaluation endpoint and should be interpreted together with the per-benchmark results.

B.4 MCP-Atlas Subset

Due to network connectivity constraints, our MCP-Atlas evaluation uses 291 of the 500 public tasks across 30 of 36 servers. The subset is determined mechanically by server availability rather than task difficulty or model performance: `mongodb`, `oxylabs`, `brave-search`, and `wikipedia` were unreachable from our evaluation network; `slack` rejected requests from the evaluation region; and `google-workspace` required unavailable OAuth verification. Every task whose reference trajectory uses any excluded server is removed. All methods in Table 1 are evaluated locally with the same judging pipeline on this identical subset. These results support within-subset comparisons, but absolute scores and rankings should not be compared directly with or assumed to transfer unchanged to the official 500-task split. Table 6 reports available task-difficulty proxies for the full and retained sets.

Table 6: Task-difficulty proxies of the retained MCP-Atlas subset (30 servers, 291 tasks) and the full public benchmark (36 servers, 500 tasks), computed from benchmark reference trajectories. The retained subset closely matches the full benchmark in reference-trajectory length and cross-server composition.

Statistic	**Full (500)**	**Kept (291)**
GT tool calls per task (mean / median / range)	4.85 / 5 / 3–17	4.74 / 4 / 3–17
Tasks with $\geq$7 GT calls	10.0%	8.2%
Multi-server tasks	99.0%	99.7%
Tasks spanning $\geq$3 servers	39.8%	37.1%

C Method Details

C.1 Topology-Aware Sampling

Our topology-aware sampling strategy performs bounded backward dependency resolution before admitting a tool. Operating on the typed dependency graph $G = (\mathcal{V}, E_{\text{flow}}, E_{\text{wf}})$ (Section 3.3.1), the algorithm proceeds in two phases for each candidate tool v:

Backward dependency resolution. Before adding v to the ordered plan τ, the algorithm recursively resolves unsatisfied inputs via `SAMPLEPRIORS`. A parameter $p_i \in \mathcal{I}(v)$ is satisfied if it is optional, user-providable per LLM classification (DeepSeek-V3.2-Chat; Appendix D), or produced by an earlier tool in τ. Each unsatisfied required parameter p_i triggers backward traversal to a producer u with a parameter-flow edge $(u \xrightarrow{p_i} v) \in E_{\text{flow}}$; workflow edges in E_{wf} are never used for resolution. Recursion depth is capped at $D_{\max}$ (Table 8), cycles are rejected, and a candidate is rejected or resampled when no producer exists, the depth bound is reached, or adding its prerequisites would exceed the plan-length budget. A stochastic override with probability ρ can add an extra prior for an already satisfiable parameter to increase diversity.

Forward expansion. Once all required inputs are resolved, the ordered prerequisite list and v are appended to τ. The algorithm then samples 1 to b unvisited successors along $E_{\text{flow}} \cup E_{\text{wf}}$ for subsequent processing. For accepted plans, the procedure guarantees dependency closure with respect to the constructed graph and parameter classifications; downstream execution-based verification validates the resulting plans and removes any remaining inconsistencies.

C.2 Simulated User

To simulate tool-use scenarios, we first classify available MCP tools into *user tools* and *assistant tools* with an LLM. User tools comprise operations that are either (i) *confidential or sensitive* (e.g., `login`, `reset_password`) or (ii) *physically constrained* (e.g., `restart_engine`). These tools require direct user authorization or physical presence and cannot be autonomously executed by the agent.

We condition the simulated user on three inputs: (a) the narrative scenario, (b) the dialogue history, and (c) the current database state. We restrict the user's knowledge to *external parameters* identified in Section 3.3.2—information that a user can realistically provide, such as personal preferences, location, or time constraints. The simulated user cannot access *internal parameters*, such as system-generated identifiers or backend state, that would be unavailable to an actual user.

C.3 Reward

Trajectory reward. For each rollout, $R_{\text{traj}} = 1$ iff every tool call of the reference trajectory is matched by a distinct rollout call; otherwise, it is 0. Matching is an injective set assignment rather than a sequence match, so equivalent call orders receive the same trajectory reward. A rollout call matches a reference call when the tool names agree and all arguments agree outside (i) the masking set from Section 3.4, which contains arguments judged not to affect correctness, and (ii) optional arguments left at their schema defaults, where omission matches an explicit default value. Extra rollout calls do not invalidate R_{traj}.

Algorithm 1 Topology-Aware Plan Sampling

Require: Typed graph $G = (\mathcal{V}, E_{\text{flow}}, E_{\text{wf}})$, maximum plan length n, start tool v_s, maximum neighbors per expansion b, depth limit $D_{\max}$
Ensure: Ordered dependency-closed plan τ with $|\tau| \le n$, or FAIL

$\tau \leftarrow [\,]$ and queue $\leftarrow$ Queue(v_s)
while queue $\neq \emptyset$ **and** $|\tau| < n$ **do**
  $v \leftarrow$ queue.dequeue()
  if $v \in \tau$ **then**
    continue
  end if
  $(P_v, s) \leftarrow \textsc{SamplePriors}(G, \tau, v, 0, D_{\max}, \emptyset)$
  if $s =$ FAIL **or** $|\tau| + |P_v| + 1 > n$ **then**
    continue ▷ reject this candidate
  end if
  $\tau \leftarrow \tau \parallel P_v \parallel [v]$
  $N(v) \leftarrow \{u \mid (v \to u) \in E_{\text{flow}} \cup E_{\text{wf}},\ u \notin \tau\}$
  if $N(v) \neq \emptyset$ **then**
    sample 1 to $\min(b, |N(v)|, n - |\tau|)$ tools from $N(v)$
    enqueue the sampled tools
  end if
end while
if $\tau = \emptyset$ **then**
  return FAIL
end if
return τ

State reward. $R_{\text{state}} \in [0, 1]$ is the per-server average of exact terminal-state matches with the state reached by the reference trajectory (binary for single-server tasks). States are compared as exact equality of each server's canonical serialized database dump. Environments are deterministic by construction (no wall-clock access; seeded randomness only), and pass@k selection fixes one canonical reference state per task. Exact comparison remains strict: semantically equivalent but distinct states (e.g., auto-assigned identifiers offset by an extra harmless call) can be under-rewarded, and read-only tasks give every rollout the same R_{state}, a constant that cancels in the group-relative advantage. These cases motivate combining the two rewards rather than treating either as a complete correctness oracle.

Length penalty. Counting redundant and malformed calls in a rollout defines

$$P_{\text{length}} = \min(0.05 \cdot n_{\text{redundant}},\ 0.5) + \min(0.05 \cdot n_{\text{malformed}},\ 0.5),$$

so the total penalty is at most 1. Malformed calls fail schema validation or raise at execution. A call is redundant if it duplicates an earlier call in the same rollout (same tool name and equal arguments outside the matching exclusions) or is flagged by the filtering criterion of Section 3.4. Harmless redundant calls, including read-only calls, leave R_{traj} and R_{state} unchanged and incur only this capped penalty.

Weights. We use $\alpha = 0.5$ and $\gamma = 1$ in all main experiments. In the BFCL sensitivity study in Section 4.4, removing either reward component reduces performance, supporting the composite design.

D Implementation Details

D.1 Data Synthesis

In the **EnvGen** pipeline of EnvFactory, we primarily use Kimi-K2-Thinking (Kimi Team et al., 2025) to propose, construct, and verify MCP environments. For **tool-graph construction**, we use DeepSeek-V3.2-Chat (DeepSeek-AI et al., 2025) in non-thinking mode to propose workflow edges and to classify tool parameters as internal or external. For **QueryGen**, we use DeepSeek-V3.2-Chat in non-thinking mode to generate RL trajectories and Qwen3-30B-A3B-Thinking-2507 (Yang et al., 2025a) to generate SFT trajectories, whose reasoning traces are distilled into the student.

Algorithm 2 Sample Priors

Require: Graph G, ordered available tools τ, current tool v, depth d, maximum depth $D_{\max}$, active recursion set A
Ensure: Ordered prior list P_v and status $s \in \{\textsc{Success}, \textsc{Fail}\}$

$P_v \leftarrow [\,]$
if $v \in A$ **then**
  return $(P_v, \textsc{Fail})$ ▷ cycle
end if
$A \leftarrow A \cup \{v\}$
for each input parameter $p_i \in \mathcal{I}(v)$ **do**
  $q \leftarrow \textsc{IsSatisfied}(p_i, \tau \parallel P_v)$
  if q and $\mathcal{U}(0,1) \geq \rho$ **then**
    continue
  end if
  $\mathcal{C} \leftarrow \{u \mid u \notin A,\ (u \xrightarrow{p_i} v) \in E_{\text{flow}}\}$ ▷ producers of p_i
  if $\mathcal{C} = \emptyset$ **then**
    if not q **then**
      return $(P_v, \textsc{Fail})$ ▷ required input has no producer
    end if
    continue
  end if
  if $d \geq D_{\max}$ **then**
    return $(P_v, \textsc{Fail})$
  end if
  $u \leftarrow \text{Uniform}(\mathcal{C})$
  if $u \notin \tau \parallel P_v$ **then**
    $(P_u, s_u) \leftarrow \textsc{SamplePriors}(G, \tau \parallel P_v, u, d+1, D_{\max}, A)$
    if $s_u = \textsc{Fail}$ **then**
      return $(P_v, \textsc{Fail})$
    end if
    $P_v \leftarrow P_v \parallel P_u \parallel [u]$
  end if
end for
if any required input of v is not satisfied by $\tau \parallel P_v$ **then**
  return $(P_v, \textsc{Fail})$
end if
return $(P_v, \textsc{Success})$

Kimi-K2-Thinking and Qwen3-30B-A3B-Thinking-2507 both decode with reasoning enabled at temperature 0.7. We generate k candidate trajectories per query before model-based selection and filtering (Table 8).

Data accounting. The reported 842 callable endpoints comprise 672 domain tools and two environment-management endpoints (`load_scenario` and `save_scenario`) for each of the 85 environments. The reported 1,622 SFT and 953 RL quantities count generated episodes: multi-turn conversations for SFT and single-request episodes for RL. Training conversion may expand one episode into multiple turn- or step-level records. We report the unit explicitly whenever comparing data scale.

D.2 Training

Supervised fine-tuning. We perform SFT using LlamaFactory (Zheng et al., 2024b) on 8×80 GB GPUs with a learning rate of 1×10^{-6} and batch size of 256, training for 3 epochs. For subsequent RL training, we initialize from the checkpoint saved after SFT. During SFT data construction, each reference tool-call or user-interaction step is converted into a separate supervision example with its associated reasoning trace and preceding dialogue/tool history. Failed tool calls remain in the history so later decisions observe their outcomes, but their output tokens are excluded from loss computation.

Table 7: Reported per-item token and elapsed-time measurements across environment construction and trajectory synthesis. Success-rate denominators and retry policies differ by stage.

Mode	Model	Prompt Token	Completion Token	Elapsed Time	Reported Success
Environment	`Kimi-K2-Thinking`	192K	31K	3 min	92.9%
SFT Trajectory	`Qwen3-30B-A3B-Thinking-2507`	228K	84K	6 min	85.4%
RL Trajectory	`DeepSeek-V3.2-Chat`	195K	19K	3 min	88.2%

Reinforcement learning. We employ Group Relative Policy Optimization (GRPO) (Shao et al., 2024) implemented with VeRL (Sheng et al., 2025). Training is conducted on 8×80 GB GPUs using a learning rate of 1×10^{-6}, rollout size of 8, and batch size of 256. We set the maximum trajectory length to 16k tokens and the maximum generation length to 4k tokens, and train for 10 epochs. Each RL episode starts from a single user request produced by QueryGen, paired with its reference tool calls and reference terminal state. The policy then interacts only with the environment, alternating tool calls and tool outputs, until it replies in natural language with a clarifying question or final answer; no simulated user is involved. The completed rollout is scored by the composite reward in Appendix C.3, and each tool-call turn is treated as an individual training sample during optimization.

D.3 Compute

For SFT data synthesis, we deploy `Qwen3-30B-A3B-Thinking-2507` on 2×80 GB GPUs. Synthesizing 1,000 multi-turn, multi-step trajectories requires approximately 20 wall-clock hours, or 40 accelerator-hours.

For SFT training, we fine-tune `Qwen3-4B` for 3 epochs using LlamaFactory (Zheng et al., 2024b) on 8×80 GB GPUs. The reported elapsed time is approximately 10 hours, corresponding to 80 accelerator-hours. For RL training, we train `Qwen3-4B` for 10 epochs using VeRL (Sheng et al., 2025) on the same hardware. The reported elapsed time is approximately 20 hours, corresponding to 160 accelerator-hours.

D.4 Token Usage

Table 7 summarizes token consumption across the pipeline. Trajectory synthesis supports asynchronous generation: synthesizing 1,000 multi-turn, multi-step trajectories takes roughly 20 hours, or approximately 1.2 minutes per trajectory.

Table 8: Hyperparameters for environment construction, topology-aware sampling, trajectory synthesis, reward computation, and post-training. Symbols in the main text refer to this table.

Name	Value	Description
Environment Construction and Tool Graph		
Parameter embedding model	BAAI/bge-m3	Tool input/output parameter embeddings.
Flow-edge similarity threshold	0.85	Cosine-similarity cutoff for proposing parameter-flow edges.
Topology-Aware Plan Sampling		
Optional-producer insertion probability ρ	0.1	Probability of adding a producer for an already satisfiable input.
Maximum backward-search depth $D_{\max}$	3	Backward producer-search recursion limit.
Maximum tools per plan n	8	Total tools in τ, including inserted prerequisites.
Maximum neighbors per expansion b	3	Unvisited outgoing neighbors sampled per admitted tool.

Continued on next page

Table 8 (continued)

Name	Value	Description
Tools per dialogue turn	1–5	Plan tools assigned to one dialogue turn.
Candidate trajectories per query k	8	Candidates generated before selection and filtering.
Reward and Model Training		
Trajectory-reward weight α	0.5	Trajectory-matching term in the composite reward R.
State-reward weight $1-\alpha$	0.5	Terminal-state term in the composite reward R.
Length-penalty weight γ	1	Scale of the redundant/malformed-call penalty.
Penalty increment and cap	0.05 per call; 0.5 per type	Per-call increment and per-violation-type cap.
SFT learning rate / batch size / epochs	1×10^{-6} / 256 / 3	LlamaFactory SFT optimization settings.
RL initialization checkpoint	after SFT epoch 1	Checkpoint used to initialize GRPO.
RL learning rate / batch size / epochs	1×10^{-6} / 256 / 10	VeRL GRPO optimization settings.
GRPO rollout group size	8	Rollouts per group-relative advantage.
Maximum RL sequence / generation length	16k / 4k tokens	Full-trajectory / generation token limits.
Synthesis and Evaluation Models		
Environment-construction model	Kimi-K2-Thinking	EnvGen proposal, implementation, and validation.
Tool-graph construction model	DeepSeek-V3.2-Chat	Workflow-edge proposal and internal/external parameter classification (non-thinking).
SFT trajectory-synthesis model	Qwen3-30B-A3B-Thinking-2507	SFT trajectories and reasoning traces.
RL trajectory-synthesis model	DeepSeek-V3.2-Chat	RL training trajectories (non-thinking).
Training hardware	8×80 GB GPUs	SFT and RL training.
Evaluation engine / tensor parallelism	SGLang / 2	Benchmark inference.
Thinking-model temperature	$T = 0.7$	Kimi-K2-Thinking, Qwen3-30B-A3B-Thinking-2507, Ours, AWM, and EnvScaler.
Non-thinking-model temperature	$T = 0$	DeepSeek-V3.2-Chat and DeepSeek-V4-Flash-Preview in non-thinking mode.
Simulated-user and judge model	DeepSeek-V4-Flash-Preview	User simulation and evaluation for τ^2-Bench and VitaBench; answer judging for MCP-Atlas.

E LIMITATIONS

ENVFACTORY uses MCP as its tool interface. Because its servers are stateful and write-capable tools can modify an environment database, each conversation uses a dedicated transport connection

to isolate its state. This design limits parallel tool invocation within a connection and creates a throughput bottleneck during large-scale data synthesis. We mitigate this bottleneck with an asynchronous pipeline that runs many isolated sessions concurrently. Future work can further characterize cross-session isolation under sustained high-concurrency workloads.

F Additional Related Work

Reinforcement Learning for LLMs. Reinforcement Learning (RL) has become a cornerstone of LLM post-training. Following the early adoption of reward-model-based pipelines (Ouyang et al., 2022), Direct Preference Optimization (Rafailov et al., 2023) streamlined this process by directly leveraging pairwise preference data. More recently, Reinforcement Learning with Verifiable Rewards (RLVR) has improved downstream performance in mathematics, coding, and agentic tasks. A prominent example is GRPO (Shao et al., 2024), which optimizes groups of sampled outputs without a separate critic. TreeRPO (Yang et al., 2025c) extends this direction with tree-sampled, step-level rewards.

Despite these advancements, the fundamental mechanics of RLVR remain under scrutiny. Notably, Yue et al. (2025) questioned whether RLVR truly expands a base model's intrinsic capabilities, demonstrating through experiments that it fails to improve Pass@k—a metric related to an LLM's reasoning coverage. Recent analyses also report declining output entropy during RLVR training and its relationship to reduced exploration (Zhu et al., 2025). To mitigate this exploration collapse, SvS (Liang et al., 2025) introduces a self-play-style problem augmentation strategy that enhances training data diversity, stabilizes entropy, and improves Pass@k performance. Alternatively, DARS (Yang et al., 2025d) combines difficulty-adaptive rollout sampling with large-batch training to improve both Pass@1 and Pass@k reasoning performance.

G Prompts

G.1 EnvGen

Tool Generation Prompt

Role. You are an expert Python Developer and MCP (Model Context Protocol) Implementation Generator. Your task is to produce a SINGLE, COMPLETE, EXECUTABLE Python file that implements class-based MCP tools using `mcp.server.fastmcp` with scenario-based state management and Pydantic schema validation.

CRITICAL OUTPUT RULES.

1. Output ONLY the final Python code wrapped in `<tool_code>` tags.
2. NO explanations, NO markdown formatting outside the tags.
3. The code must be production-ready, strictly following the 4-Section structure.

Implementation Architecture.

1. File Structure (Mandatory).

- **Section 1: Schema**: Pydantic models (Entity models + 1 Scenario model). `Scenario_Schema` defines the internal state structure of the Class.
- **Section 2: Class**: Main logic class.
- **Section 3: MCP Tools**: FastMCP registration + Wrappers.
- **Section 4: Entry Point**: `mcp.run()`.

2. Core Requirements.

2.1. Pydantic Models.

- Use Pydantic v2 API throughout. Do NOT use deprecated v1 patterns, such as `.dict()`. Use `model_dump()` instead.
- Define all data structures using Pydantic `BaseModel` classes.
- Import: `from pydantic import BaseModel, Field` and `from typing import Dict, List, Optional, Union, Any`.
- Each model must inherit from `BaseModel`, use `Field()` with descriptions/defaults, include type hints, and have docstrings.

- Entity model naming rule: Entity model class names and field names MUST NOT start with underscore. For example, use `Item` not `_Item` for class name, and use `id` not `_id` for field name.
- Simplify Nested Structures: For fields in the Scenario model that store complex nested dictionaries or variable schemas, such as configuration maps, weather patterns, or lookup tables, **use `Dict[str, Any]` or `dict`**. Do NOT use strict complex recursive types, such as `Dict[str, Dict[str, Union[str, float]]]`, to ensure robustness during scenario loading.
- Create individual entity models and one main scenario model defining the complete scenario structure.
- External data tables, also known as reference lookup data, should be defined directly as fields in the Scenario model using `Field(default={{...}})` or `Field(default_factory=lambda: {{...}})` with 10–20 entries.
- Current Time Management: All references to "current time", "now", or "current date/time" MUST be stored as string fields in the Scenario model. For example:

```
current_time: str = Field(
    ...,
    pattern=r"^\d{4}-\d{2}-\d{2}T\d{2}:\d{2}:\d{2}$",
    description="Current timestamp in ISO 8601 format"
)
```

- NEVER use `datetime.now()` or any real-time functions. Access current time through scenario state variables, such as `self.current_time`.
- End with `Scenario_Schema = [Model1, Model2, ScenarioModel]` listing all Pydantic classes.
- `Scenario_Schema` represents the internal state structure of the Class instance.

Pydantic Model Validation. You MUST use Pydantic `Field` constraints to enforce data validation at the model level. This is the PRIMARY validation mechanism.

- **Pattern Validation**: Use `Field(..., pattern=r"pattern")` for string fields with format requirements, such as codes, IDs, times, and dates. **IMPORTANT: Always use raw strings (`r""`) for regex patterns.**
  - Time fields: `pattern=r"^([0-1]?[0-9]|2[0-3]):[0-5][0-9]$"` for HH:MM format.
  - ISO 8601: `pattern=r"^\\d{4}-\\d{2}-\\d{2}T\\d{2}:\\d{2}:\\d{2}$"` for timestamps, and `pattern=r"^\\d{4}-\\d{2}-\\d{2}$"` for dates.
- **Range Constraints**: Use `Field(..., ge=0)` for non-negative numeric fields, `Field(..., ge=0, le=100)` for ranges, and `Field(..., gt=0)` for strictly positive values.
- **Validation Philosophy**: Prefer `Field` constraints for ALL format/range validation. Use `@field_validator` or `@model_validator` only when `Field` constraints are insufficient, such as cross-field validation. Pydantic v2 automatically validates when data is loaded via `load_scenario`.

2.2. Implementation Pattern.

- Create a Python class containing all MCP tools as public methods.
- Private methods, starting with _, are helpers and not registered as MCP tools.
- After the class, create FastMCP instance and class instance.
- Register each public method as MCP tool using `@mcp.tool()` decorator with wrapper functions.
- Method signatures must exactly match `input_schema` properties with correct types and names.
- Return values must precisely match `output_schema` structure.
- **MCP tool wrapper function return type annotations should use simple types**, such as `-> str` or `-> dict`, NOT specific Pydantic model types. For example, avoid `-> Tweet`; use `-> dict` instead.

2.3. Required Methods.

2.3.1. `__INIT__`

- Initialize all state variables as class attributes with type hints.
- Do not set default values. They come from scenario loading.

2.3.2. `LOAD_SCENARIO` (REQUIRED)

- Signature: `def load_scenario(self, scenario: dict) -> None:`
- Instantiate main scenario Pydantic model. For example, `scenario_model = ScenarioModel(**scenario)`.
- Assign validated fields to class attributes. For example, `self.field = scenario_model.field`.
- Pydantic handles type conversion automatically.
- Return `None` on success, with empty `output_schema`.

2.3.3. `SAVE_SCENARIO` (REQUIRED)

- Signature: `def save_scenario(self) -> dict:`
- Return dictionary containing all current state variables.
- Serialize Pydantic model instances using `model_dump()` in Pydantic v2. Do NOT use `.dict()`, which is deprecated.
- The returned dictionary structure must exactly match the structure expected by `load_scenario`, with the same field names and types.

2.4. State Management.

- **State as Truth**: Class instance holds all state. State variables, such as `self.xxx`, serve as internal database tables/collections. All tools must operate directly on state variables. NEVER simulate external APIs, network requests, or create fake/mock data.
- **Scenario Loading**: Convert `dict -> Pydantic Model -> self.variables`. Pydantic models define structure; state variables store actual data as dicts/lists.
- **Anti-Lazy Logic**: Lookup tools MUST query the reference data from Schema fields, such as `self.xxxMap`. NEVER return hardcoded values like `return 100`. Access reference data through class attributes that correspond to Schema fields, such as `self.taxRatesMap`, not hardcoded values.
- **Time Management**: When tools need to access "current time" or "now", they MUST read from scenario state variables, such as `self.current_time` or `self.current_date`. NEVER use `datetime.now()`, `time.time()`, or any real-time functions. All time values are provided through the scenario and stored as strings.
- Perform CRUD operations on state variables: READ, WRITE, CREATE, UPDATE, and DELETE. Handle missing data: return empty results for reads and error dicts for operations requiring existing data. NEVER create fake data to fill responses.

2.5. Reference Data Management.

STATIC REFERENCE DATA PATTERN (FOR LOOKUP TOOLS)

1. **Scenario Model**: Add reference data fields directly to the Scenario model as ordinary Pydantic fields. For example, `taxRatesMap: Dict[str, float] = Field(default={{...}}, description="...")`. Use `Field(default={{...}})` or `Field(default_factory=lambda: {{...}})` to set default values containing 10–20 entries. These fields are just like other Scenario fields, with no special handling needed.
2. **Class Init**: Initialize corresponding class attributes. For example, `self.taxRatesMap: Dict[str, float] = {{}}`.
3. **Load Scenario**: Pydantic automatically handles default values. If scenario provides the field, use the provided value; otherwise, use the default value from `Field(default=...)`.
4. **Tool Methods**: Access reference data directly through class attributes, such as `self.taxRatesMap`. NEVER return hardcoded values.
5. **Save Scenario**: Return all fields in the dictionary, including reference data fields.

2.6. Random Number Generation (Reproducibility).

- **Avoid random when possible**: Prefer deterministic logic based on state variables or input parameters. If random is necessary, add `random_seed: Optional[int] = Field(default=None, description="Random seed for reproducible results")` to the Scenario model, initialize `self.random_seed` in `__init__`, and use `random.seed(self.random_seed)` in methods that need randomness.

3. MCP Tools.

3.1. Error Handling & Empty Output.

- **Class methods**: MUST NOT contain try-except blocks or error detection logic. Directly perform operations. Return normal results or `None`, for empty output. Let exceptions propagate naturally.
- **MCP wrapper functions**: MUST use try-except blocks for all error detection and handling.
  - **Simplified validation**: MCP wrapper functions should perform ONLY basic parameter existence checks, such as non-empty and non-None, and type checks using `isinstance`.
  - **DO NOT duplicate format/range validation**: Rely on Pydantic model validation when data passes through `load_scenario` or when creating model instances.
  - **Business logic checks**: Perform ID existence checks, state-dependent validations, such as "item not found" or "insufficient balance", and other business logic validations.
  - If class method returns `None`, meaning empty output, return success message string with `-> str`. Otherwise return result directly with `-> dict`.

Validation Responsibility Division:

- **Pydantic Models (Primary)**: Handle ALL format validation, including pattern validation, range validation using `ge`, `gt`, `le`, and `lt`, and type conversion.
- **MCP Tool Wrappers (Secondary)**: Handle parameter existence, such as non-empty values, basic type checks using `isinstance`, and business logic validations, such as ID existence and state checks.
- When data is loaded via `load_scenario(scenario: dict)`, Pydantic automatically validates all fields. There is no need to re-validate format/ranges in tool wrappers.

3.2. Docstring Requirements (CRITICAL).

- **Class methods (Section 2)**: Only require a single-line docstring describing what the method does.
- **MCP tool wrapper functions (Section 3)**: MUST have complete Google-style docstring with three sections:
  - **Description**: What the method does.
  - **Args**: All parameters with types and descriptions. Mark optional with `[Optional]`.
  - **Returns**: All return fields with types and descriptions, matching `output_schema`.

Reference Implementation.

`{MCPToolGenerator_Example}`

Task. Generate the MCP tool code based on the user's specific requirement. Output ONLY the code inside `<tool_code>...</tool_code>`.

Test Cases Generation Prompt

Role. You are an expert test scenario generator for MCP tool implementations. Your goal is to create diverse, comprehensive test scenarios that thoroughly validate tool functionality.

Responsibilities.

1. Analyze Tool Code Structure.

- Examine the provided `tool_code` to identify the main Pydantic scenario model, such as `GoogleCalendarScenario`, `TwitterScenario`, or `InventoryScenario`.
- Understand all fields, their types, default values, and relationships.
- Identify reference data fields, such as lookup tables like `taxRatesMap` and `shippingZonesMap`.
- Understand the tool methods and their expected behaviors.

1.5 Pydantic Model Type Matching (CRITICAL). Before generating scenario data, you MUST carefully analyze the Pydantic model field types to ensure exact type matching.

Complex Type Patterns.

- **`Dict[str, BaseModel]`**: Generate:

```
{"key": {"field1": value1, "field2": value2, ...}}
```

Example:

```
tickets: Dict[str, TicketInfo]
```

Should become:

```
{"T001": {"price": 100.0, "seats": 50}}
```

WRONG:

```
{"T001": "ticket_info"}
```

or:

```
{"T001": "some_string"}
```

- **`List[BaseModel]`**: Generate:

```
[
  {"field1": value1, ...},
  {"field1": value2, ...}
]
```

Example:

```
routes: List[RouteInfo]
```

Should become:

```
[
  {"from": "BJ", "to": "SH"},
  {"from": "SH", "to": "GZ"}
]
```

WRONG:

```
["route1", "route2"]
```

or:

```
[{"name": "route1"}]
```

- **Nested BaseModel classes**: Look for class definitions in the code.

For example, if you see:

```
class TicketInfo(BaseModel):
    price: float
    availability: int
```

Then:

```
Dict[str, TicketInfo]
```

expects:

```
{"key": {"price": 100.0, "availability": 50}}
```

NOT:

```
{"key": {"ticket_id": "T001"}}
```

because that uses wrong fields.

- **Type consistency**: Ensure value types match exactly.
  - `price: float` → Use `100.0` as a float, NOT `"100"` as a string. `100` as an integer is acceptable, but float is better.
  - `count: int` → Use `50` as an integer, NOT `"50"` as a string.
  - `available: bool` → Use `true/false` as boolean values, NOT `"true"` as a string.

Validation Rules.

1. Read ALL Pydantic class definitions in Section 1 (Schema) of the `tool_code`.
2. Map each field in the main Scenario model to its actual type.
3. For complex types, such as `Dict` or `List` with `BaseModel`, identify the nested structure.
4. Generate data that exactly matches the nested structure.
5. DO NOT guess or simplify complex types. Match them precisely.

2. Generate Diverse Test Scenarios. You must generate **`{n_scenarios}`** test scenarios with varying complexity levels.

Complexity Levels.

1. **Simple (1–2 scenarios)**: Minimal data.
   - 1–2 main entities, such as 1 calendar with 1 event, or 2 items in inventory.
   - Basic fields populated.
   - Use default reference data if applicable.
   - Purpose: Test basic tool functionality.
2. **Medium (2–3 scenarios)**: Moderate data.
   - 3–5 main entities with varied attributes.
   - Mix of populated and empty optional fields.
   - Some edge cases, such as events at midnight or items with zero price.
   - Purpose: Test typical use cases.
3. **Complex (1–2 scenarios)**: Rich data.
   - 5–10 main entities with diverse relationships.
   - All fields populated with realistic values.
   - Nested structures fully used.
   - Purpose: Test scalability and complex interactions.
   - **IMPORTANT**: Focus on functional coverage, not data volume. Use representative data samples rather than exhaustive datasets to avoid JSON serialization/deserialization issues.
4. **Boundary (as needed)**: Edge cases.
   - Empty collections, such as no calendars or no items. These should pass.
   - Extreme values, such as very long strings or max integers. These should pass or fail appropriately.
   - **Invalid inputs**, such as special characters violating validation rules. These should be rejected.
   - Purpose: Test error handling and edge cases.

IMPORTANT: For boundary scenarios that test invalid inputs, you MUST specify `expected_behavior`:
- `"pass"`: Normal success expected, such as empty collections or extreme but valid values.
- `"validation_error"`: Tool should reject input with validation error, such as special characters violating a pattern.
- If not specified, defaults to `"pass"`.

3. Ensure Scenario Quality. Each scenario must:
- Be a complete, valid dictionary matching the scenario model structure.
- Include ALL required fields from the Pydantic model.
- Use realistic, coherent data, such as consistent date ranges and related IDs.
- Have unique identifiers, such as different event IDs or item IDs.
- Include reference data fields with their default values, or variations if testing lookup functionality.
- For boundary scenarios with invalid data, include `expected_behavior: "validation_error"` to indicate expected rejection.
- If the scenario model includes `random_seed`, provide a fixed integer value, such as `42`, to ensure reproducible results.

Data Volume Guidelines:
- Keep scenario data concise and manageable to avoid JSON serialization/deserialization errors.
- For tools with large datasets, such as train schedules or route maps, use small but representative samples, usually 2–5 entries, rather than exhaustive data.
- Complex nested structures should be simplified. Test functionality, not data volume.
- If a tool involves lookup tables or reference data, include only essential entries needed for testing, typically 3–10 entries.

4. Output Format. Your response must strictly follow this structure:

```
<scenarios>
[
  {
    "scenario_id": "scenario_001",
    "complexity_level": "simple",
    "description": "Brief description of what this scenario tests",
    "expected_behavior": "pass",
```

```
    "scenario_data": {
      // Complete scenario dictionary matching the Pydantic model
    }
  },
  {
    "scenario_id": "scenario_002",
    "complexity_level": "medium",
    "description": "Brief description of what this scenario tests",
    "expected_behavior": "pass",
    "scenario_data": {
      // Complete scenario dictionary matching the Pydantic model
    }
  },
  {
    "scenario_id": "scenario_005",
    "complexity_level": "boundary",
    "description": "Test invalid data with special characters (should be rejected)",
    "expected_behavior": "validation_error",
    "scenario_data": {
      // Scenario with intentionally invalid data
    }
  }
  // ... more scenarios up to {n_scenarios}
]
</scenarios>
```

Important Notes.

- Each `scenario_id` must be unique, such as `"scenario_001"` or `"scenario_002"`.
- `complexity_level` must be one of: `"simple"`, `"medium"`, `"complex"`, or `"boundary"`.
- `expected_behavior` must be `"pass"` by default or `"validation_error"` for scenarios testing invalid input rejection.
- `scenario_data` must be a complete, valid scenario dictionary.
- Include variety in your test data to maximize test coverage.

Validation Prompt

Role. You are a comprehensive MCP tool validator. Your task is to validate a single test scenario by executing all available tools and diagnosing any issues.

Responsibilities.

1. Scenario Preparation. You will receive:

- `mcp_server_name`: Name of the MCP server
- `tool_code`: MCP Tools section (Section 3) containing FastMCP registration and tool wrapper functions
- `tools_metadata`: List of all available tools with their schemas
- `scenario_id`: Unique identifier for this scenario
- `scenario_data`: The test scenario data
- `request_id`: For constructing `client_id`

2. Client ID Construction. You must use this exact pattern:

- `client_id = "{mcp_server_name}-{request_id}_{scenario_id}"`
- Example: `"GoogleMaps-abc123_scenario_001"`
- Use the SAME `client_id` for all operations in this scenario

3. Understanding Expected Behavior. The scenario may include an `expected_behavior` field:

- **`"pass"`** (default): Normal execution, tools should succeed
- **`"validation_error"`**: Scenario contains invalid data, tools should reject it with validation error

When evaluating results:

1. **Pass**: Tool executed successfully with expected output when `expected_behavior="pass"`.
2. **Expected Failure**: Tool correctly rejected invalid input with validation error when `expected_behavior="validation_error"`.
   - THIS COUNTS AS PASSED. The tool is working correctly by rejecting bad data.
3. **Unexpected Failure**:
   - Tool raised error when success was expected, meaning `expected_behavior="pass"` but got error.
   - OR: Tool succeeded when validation error was expected, meaning `expected_behavior="validation_error"` but no error.

4. Layered Validation Procedure.

Layer 1: Scenario Loading (Critical and Blocking). Call `execute_mcp_tool` with:

- `tool_name`: "{mcp_server_name}-load_scenario"
- `tool_args`: {"scenario": scenario_data}
- `client_id`: as constructed above

Record the result.

Evaluate based on `expected_behavior`:

- If `expected_behavior="validation_error"` and `load_scenario` fails with validation error: PASS, as an expected failure.
- If `expected_behavior="pass"` and `load_scenario` succeeds: PASS.
- Otherwise: FAIL, as unexpected behavior.

IMPORTANT: If `load_scenario` FAILS unexpectedly:

- Mark it as CRITICAL error in the errors list.
- **STOP validation immediately and return**. Do not proceed to test other tools.
- This is a blocking failure that prevents further validation.

Layer 2: Tool Execution (Conditional). Only execute if `load_scenario` succeeded:

For each tool in `tools_metadata`, excluding `load_scenario` and `save_scenario`:

- Use the loaded scenario state.
- Design 2–3 test cases with different inputs:
  - **Valid case**: Normal, expected inputs
  - **Boundary case**: Edge values, if applicable
  - **Error case**: Invalid inputs, if error handling should be tested
- For each test case:
  - Call `execute_mcp_tool` with the tool and test inputs
  - Record: input, expected behavior, actual output, and any errors
  - Evaluate: Does output match expected? Are there any unexpected errors?
- This layer helps identify tool logic errors independent of scenario loading.

Layer 3: State Consistency (Conditional).

- Only run this if `load_scenario` succeeded.
- After executing all tools, call:
  - `tool_name`: "{mcp_server_name}-save_scenario"
  - `tool_args`: {{}}
  - `client_id`: same as before
- Record the saved scenario.
- Compare with the original scenario + expected modifications.
- If `load_scenario` failed, skip this step with note `"Skipped due to load_scenario failure"`.

4. Error Diagnosis. For any failures, provide:

- **Error type**: For example, `"Tool execution error"`, `"State inconsistency"`, or `"Schema mismatch"`
- **Error location**: Which tool/method failed
- **Error details**: Actual error message, stack trace if available
- **Expected vs Actual**: What was expected vs what happened

- **Root cause analysis**: Why did this fail? For example, "load_scenario does not handle empty lists" or "tool returns wrong field name"

6. Output Format. Your response must strictly follow:

```
<validation_result>
{
  "scenario_id": "...",
  "passed": true/false,
  "load_scenario_result": {
    "success": true/false,
    "error": "..."  // provide the error message if failed
  },
  "tool_execution_results": [
    {
      "tool_name": "...",
      "passed": true/false,
      "error": "..."  // provide the error message if failed
    }
  ],
  "save_scenario_result": {
    "success": true/false,
    "consistency_check": true/false,
    "error": "..."  // provide the error message if failed
  },
  "errors": [
    {
      "error_type": "...",
      "error_location": "...",
      "error_details": "...",
      "expected_vs_actual": "...",
      "root_cause": "...",
      "expected_error": true/false
    }
  ]
}
</validation_result>
```

Important Notes.

- Test ALL tools, except `load_scenario` and `save_scenario`.
- Use the SAME `client_id` throughout.
- Properly classify `result_type` based on `expected_behavior` from the scenario.
- For `expected_behavior="validation_error"`, set `expected_error=true` when validation error occurs.
- Provide detailed error diagnosis for UNEXPECTED failures only.
- Even if one tool fails, continue testing other tools.

Tool Revise Prompt

Role. You are an expert MCP tool code reviser. Your task is to analyze validation failures from multiple test scenarios and fix all issues systematically.

Core Responsibilities.

1. Error Categorization and Scenario Problem Detection. Categorize errors into:

- **Pydantic Model Issues**: Schema definition problems, field type mismatches
- **Load/Save Scenario Issues**: State management problems, missing fields in save
- **Tool Logic Errors**: Incorrect implementation, wrong return values, missing error handling
- **State Management Issues**: Tools not reading/writing state correctly, state inconsistencies
- **Schema Mismatches**: Input/output does not match declared schemas

IMPORTANT: Distinguish between Code Problems and Scenario Problems

Before fixing code, you must determine if the failures are due to:
- **Scenario Problems** (`is_scenario_problem=true`): Failures caused by invalid or poorly designed test scenarios
  - Scenario data does not match tool schema, such as missing fields, wrong types, or values out of range
  - Scenario data has logical errors, such as references to non-existent IDs or inconsistent data
  - Scenario's `expected_behavior` is incorrectly set, such as should be `"pass"` but marked as `"validation_error"`, or vice versa
  - Scenario tests non-existent functionality or tools
  - Scenario violates tool's documented constraints or requirements
- **Code Problems** (`is_scenario_problem=false`): Failures caused by tool implementation issues
  - Tool logic errors in implementation
  - Incomplete or incorrect schema definitions
  - State management problems
  - Missing error handling
  - Tools not following MCP requirements

Judgment Principle: If errors are due to scenario data not meeting tool requirements or poor scenario design, set `is_scenario_problem=true`. If errors are due to tool code implementation issues, set `is_scenario_problem=false`.

2. Prioritized Fix Strategy. The errors are categorized by severity. Fix issues in this order:
1. **CRITICAL (Must Fix First)**:
   - `load_scenario` failures, since these block all testing
   - Pydantic model schema mismatches, including validation errors and type mismatches
   - These affect all scenarios and must be fixed before anything else
2. **HIGH (Fix Next)**:
   - Tools that fail in multiple scenarios
   - Tool logic errors affecting core functionality
   - State management issues

IMPORTANT: If the error severity summary shows CRITICAL errors with count greater than 0, you MUST fix those first before addressing other issues. Do not make changes to working code until critical issues are resolved.

3. Fix Implementation Guidelines.
- Fix the root cause, not symptoms
- Ensure fixes do not break currently passing scenarios
- Maintain all original functionality and structure
- Follow all MCP tool generation requirements
- Test edge cases in your mental model before suggesting fixes

When fixing, verify:
- All Pydantic models are complete and correct
- `load_scenario` properly validates and assigns all fields
- `save_scenario` returns all current state fields
- Tool methods correctly access state via `self.xxx`
- Return values exactly match output schemas
- Error handling for missing data and invalid inputs
- Reference data fields are properly initialized and used

G.2 TOOLGRAPH

Logical Refinement Prompt for ToolGraph

Role. You are an expert tool relationship analyst specializing in dependency inference. Your task is to **augment** the current tool dependency graph by adding *only missing, justified directed edges*.

You are given:

1. **Tool Descriptions**: A list of tools, each with name, functional description, and parameters.

2. **Current Adjacency Map**: A dict `tool_name → [list of successor tool names]`, representing *existing* dependencies (Tool A → Tool B means Tool B may depend on or follow Tool A).

Guidelines. For every *candidate* ordered pair (Tool A → Tool B) **not already present**, assess:
- **Semantic Complementarity**: Do the tools solve parts of a shared task or pipeline? (e.g., preprocessing → analysis)
- **Data Flow Feasibility**: Can outputs (explicit, implicit, or inferred context) from Tool A reasonably inform or enable Tool B's execution?
- **Workflow Plausibility**: Would a rational user *naturally* run Tool B after Tool A in a realistic scenario?
- **Parameter/Context Alignment**: Are parameters, domains, or expected inputs/outputs conceptually aligned—even if naming differs?

Constraints.
- No self-loops (Tool A → Tool A is forbidden).
- Only add edges where Tool A and Tool B are distinct and exist in the tool list.

G.3 QUERYGEN

Simulated User Prompt

Role. You are a realistic human user interacting naturally with an assistant.

Here is what you know.
- **Scenario and User Profile:**
  `{scenario}`
- **User Intent:**
  `{user_intent}`
- **Hidden MCP Servers:**
  `{mcp_server_config}`

Here are the available tools you can use. You have access to the following tools within `<tools></tools>` XML tags:

```
<tools>
{mcp_server_tools}
</tools>
```

Conversation.

`{conversation}`

Guidelines. Your response should follow these guidelines step-by-step:
1. **Natural Voice**
   - Speak in first-person as a real person would speak. Respond conversationally and naturally.
2. **Task Scope**
   - Focus only on the current turn. When the assistant asks if you have anything else, do not invent new requests.
3. **Knowledge Boundaries**
   - Only share knowledge a real person would realistically recall, as indicated in your user knowledge.
4. **Don't Over-Help**
   - If the information the assistant is asking for is already provided in previous conversation or should be discovered through tools or reasoning, do not directly provide. Instead, hint to the assistant how to get it.
   - Don't describe what you'll do. Directly and concisely respond to the assistant's question.
5. **Tool-Use Discipline**
   - When the assistant gives you a direct, actionable instruction, you may use tools:
     – "Can you login to the website with your account?"

  - "Try restarting the car engine."
  - "Please turn on your mobile phone."
- Only use tools when explicitly instructed and actionable. Never use tools for inquiry or general questions.
- You can only use tools from the "Available User Tools" section.
- After using tools, you need to respond to the assistant accordingly.

Response Format. For each function call, return a JSON object with function name and arguments within <tool_call></tool_call> XML tags:

```
<tool_call>
{"name": <function-name>, "arguments": <args-json-object>}
</tool_call>
```

Otherwise, respond to the assistant directly. Please check each guideline before responding. Please avoid calling tools and responding to the assistant in the same step.

Assistant Prompt

Role. You are a helpful assistant. Your goal is to fulfill the user's requests in an interactive environment.
At each step, you will receive either the user's request/reply or the tool call results.

- If you can proceed with the current information, select proper tools from the tool set and provide complete, valid parameters.
- If you lack essential information to complete the task or perform a tool call, and it cannot be obtained through the existing tool set, actively ask the user for specific details.
- Avoid calling tools while interacting with user in one step.
- When a task involves sensitive credentials or physical device actions (e.g., logging into an account or restarting a phone), provide explicit step-by-step instructions naming the specific tools and required parameters.
- You cannot execute user tools directly; instead, guide users on how to perform these actions themselves.
- When you believe the task is completed, provide a direct and concise response to the user's original request.

Here are the actions you may instruct the user to do:

```
{user_tools}
```

Conversation.

```
{conversation}
```

Tools. You may call one or more functions to assist with the user query.
You are provided with function signatures within <tools></tools> XML tags:

```
<tools>
{tools}
</tools>
```

For each function call, return a JSON object with function name and arguments within <tool_call></tool_call> XML tags:

```
<tool_call>
{"name": <function-name>, "arguments": <args-json-object>}
</tool_call>
```

Scenario Planner Prompt

Role. You are a scenario planner specialized in creating high-quality initial contextual scenario for multi-turn conversation. You will be given:
1. **Tool Call Trace:** A list of tools (from various MCP servers) that appeared in a conversation.

Instruction.

1. Scenario Design. Design a cohesive narrative that *naturally motivates* the observed tool sequence. Your scenario must:

- Define a realistic user persona (name, age range, occupation, location, relevant traits)
- Establish a concrete situation with time/place/context that explains *why* the user would perform these actions
- Flow logically from initial need → actions taken → implied next steps
- **Never mention tools, APIs, or technical mechanisms**—describe only human behaviors and motivations
- Be specific and grounded (avoid generic phrases like "a user wanted information")
- Reflect cultural and situational plausibility for the geographic/occupational context

Query Generation Prompt

Role. You are a simulated user. Your task is to generate the most plausible, natural user request that would directly and exclusively motivate the target tool call(s) in the current turn.

Guidelines.

Clarity & Naturalness.
- Be conversational and realistic. Avoid robotic phrasing, checklists, overly technical jargon, or specific tool name and parameter.
- Speak strictly in the first-person perspective.
- Build logically on prior context using natural references (e.g., "the hotel you found earlier", "since we're sticking to that budget").

Background Analysis.
- Analyze what previous turns accomplished.
- Use the scenario and user profile to shape tone, preferences, and constraints (e.g., budget-conscious, eco-friendly).

Target Tool Analysis.
- Analyze the provided target tool calls to identify their underlying subgoals.
- Determine the logical relationship between these subgoals (sequential, parallel, or conditional).
- Weave them into a single, cohesive natural language query that naturally motivates all tool executions.
- Ensure smooth transitions and logical flow, concatenating independent subgoals where appropriate.

User Intent.
- Firstly briefly analyze what previous turns achieve, then explain the user intent for this turn, including the goals, constraints (e.g. tight budget), and the preferences (e.g. cheaper ticket).