EnvFactory: Scaling Tool-Use Agents via Executable Environments Synthesis and Robust RL

Minrui Xu[1,*], **Zilin Wang**[1,*], **Mengyi Deng**[1], **Zhiwei Li**[1], **Zhicheng Yang**[1], **Xiao Zhu**[1], **Yinhong Liu**[3], **Boyu Zhu**[4], **Baiyu Huang**[1], **Chao Chen**[1], **Heyuan Deng**[2], **Fei Mi**[2], **Lifeng Shang**[2], **Xingshan Zeng**[2,†] **and Zhijiang Guo**[1,5†]

[1]LARK, HKUST (GZ), [2]Huawei Technologies Co., Ltd, [3]University of Cambridge, [4]UCL, [5]HKUST

[*]Equal Contribution [†]Corresponding Author

Github Page: https://github.com/LARK-AI-Lab/EnvFactory

Abstract

Equipping LLMs with tool-use capabilities via 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 captures implicit human reasoning. Existing approaches depend on costly real-world APIs, hallucination-prone LLM simulators, or synthetic environments that are often single-turn or depend on pre-collected documents. Moreover, synthetic trajectories are frequently over-specified, resembling instruction sequences rather than natural human intents, reducing their effectiveness for RL training. 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 calibrated refinement, producing grounded queries with implicit intents. Using only 85 verified environments across 7 domains, EnvFactory generates 2,575 SFT and RL trajectories. Despite using significantly fewer environments than prior work, which are often 5 times more, EnvFactory achieves superior training efficiency and downstream performance, improving Qwen3-series models by up to +15% on BFCLv3, +8.6% on MCP-Atlas, and +6% on conversational benchmarks including τ^2-Bench and VitaBench. By fully automating both environment construction and trajectory synthesis, EnvFactory provides a scalable, extensible, and robust foundation for Agentic RL.

1. Introduction

Equipping Large Language Models (LLMs) with tool-use capabilities has significantly expanded the frontier of AI agents (Luo et al., 2025; Qu et al., 2025). Interacting with external tools enables real-time information retrieval, precise computation, and complex system orchestration. Early approaches (Xu et al., 2025; Yang et al., 2025b) typically rely on *supervised fine-tuning* (SFT) to teach tool-calling formats and interaction patterns, while more work explores *agentic reinforcement learning* (Agentic RL), where agents acquire tool-use policies through trial-and-error interactions with users and executable environments (Feng et al., 2025; Hao et al., 2025; Jin et al., 2025). Such frameworks typically involve three key components: agents, environments, and users. The interplay between these components is critical for learning effective tool-use abilities.

The effectiveness of Agentic RL ultimately hinges on two core factors: **environments** and **data**. Scalable and executable environments must faithfully capture real-world interaction dynamics while

Corresponding author(s): Zhijiang Guo(zhijiangguo@hkust-gz.edu.cn), Xingshan Zeng(zeng.xingshan@huawei.com)

ensuring low-latency and stable execution. Meanwhile, realistic and verified tool-use data, which reflects contextual ambiguity and implicit reasoning, are essential for improving generalization and providing reliable reward signals for stable policy optimization.

However, existing approaches fall short on either fronts. From the environment perspective, prior methods generally fall into three categories. (1) *Production environments* (Guo et al., 2025; Hao et al., 2026; Qin et al., 2023; Xu et al., 2025), such as real-world APIs or MCPs, provide authentic execution, but remain costly to scale and destabilize RL training due to potential network latency. (2) *Simulated environments* (Chen et al., 2025c; Li et al., 2025, 2026b) use LLMs to emulate tool behavior, enabling rapid prototyping but often suffering from hallucination, which makes RL training difficult to generalize in real-world application (Kalai et al., 2025; Wang et al., 2024). (3) *Synthetic environments* reconstruct tools through sandboxed code, offering a balance between realism and scalability (Cai et al., 2025; Fang et al., 2025). However, existing synthetic methods exhibit several key limitations: some approaches rely solely on stateless environments (Sullivan et al., 2025; Ye et al., 2026), while others depend on pre-collected documents, which limits their generalization to unseen tool ecosystems (Cai et al., 2025; Fang et al., 2025).

Another gap exists on the data side. In real-world, user requests are often concise and implicit, requiring agents to perform logical inference and contextual reasoning. Capturing such interaction patterns is crucial, as they faithfully reflect real-world usage while introducing richer decision-making challenges for agent training. However existing synthetic trajectories are commonly over-specified to ensure pass rate, explicitly enumerating task requirements and reasoning steps (Xu et al., 2025; Yin et al., 2025). Consequently, these trajectories resemble rigid "instruction lists" rather than natural human intents, limiting both their realism and value for training agentic decision-making.

To address these limitations, we propose **EnvFactory**, a fully automated framework that unifies robust environment construction and realistic trajectory generation with topology-aware graph-based guidance. At the environment level, EnvFactory autonomously proposes diverse tool-use scenarios and explores authentic online resources, enabling scalable expansion to previously unseen tool ecosystems while preserving strong fidelity to real-world usage. Based on these structured proposals, EnvFactory automatically constructs stateful databases and executable tool interfaces, followed by rigorous verification and iterative refinement to ensure robustness. This fully automated pipeline enables the scalable creation of diverse, low-latency, and reliable environments for Agentic RL.

At the data level, EnvFactory addresses the realism gap in existing synthetic trajectories by two strategies: First, a topology-aware sampling strategy recursively resolves logical dependencies during sampling, ensuring that the guided tools form a coherent logical foundation for query generation. Second, a calibrated refining stage injects realistic human communication patterns—including implicit intents and ambiguity—into the generated queries, transforming the rigid "instruction lists" into natural human requests.

Using EnvFactory, we construct 85 verified environments comprising 842 tools across diverse domains, including *commerce, finance, travel, office, lifestyle, research, and utilities*, as illustrated in Figure 1. Building on these environments, we synthesize 1,622 SFT and 953 RL multi-turn, multi-step trajectories for post-training. Despite using significantly fewer environments than concurrent work (Song et al., 2026; Wang et al., 2026), which are often 5 times more, EnvFactory achieves higher training efficiency and stronger downstream performance, improving Qwen3-series models by up to **15%** on BFCLv3, **8.6%** on the real-world MCP benchmark MCP-Atlas, and **6%** on conversational benchmarks, including τ^2-Bench and VitaBench. We summarize our contributions as follow:

- We propose **EnvFactory**, a unified autonomous pipeline for scaling diverse, executable tool environments and synthesizing realistic, verified trajectories for both SFT and RL training.

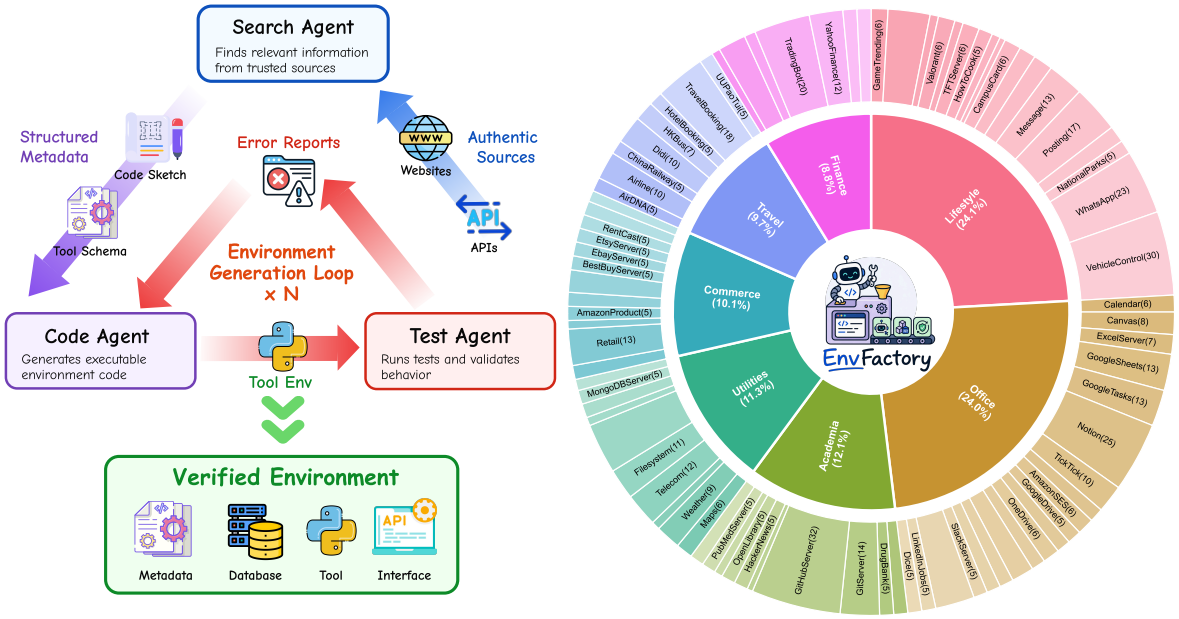

Figure 1 | The left figure presents an overview of *EnvGen*: the Search Agent autonomously proposes and searches for authentic sources; the Code Agent implements the database and code using feedback from the Test Agent; and the Test Agent generates test cases and error reports. The collaboration between three agents construct diverse, verified environments. The right figure displays a sunburst plot of environments , with the inner ring indicating the proportion of each domain they belongs to and the outer ring showing the number of tools for each environment.

- We introduce a novel topology-aware sampling algorithm that recursively resolves tool dependencies and synthesizes coherent, natural multi-turn trajectories with implicit intents.
- Extensive experiments highlight the data efficiency of ENVFACTORY and its effectiveness for training agents in complex tool-use environments.

2. Related Work

Environment Scaling for Tool Agents. The tool-augmented LLM agents is deeply tied to the quality of environments. Existing environment construction strategies fall into three paradigms. **Production environments** employ real-world APIs (Qin et al., 2023) and MCP servers (Xu et al., 2025) to provide authentic execution. However, they are expensive to scale and suffer from network latency, which destabilizes RL training. **Simulated environments** leverage LLMs to emulate tool behavior and state dynamics, enabling rapid prototyping (Chen et al., 2025c; Li et al., 2025, 2026b). However, they are prone to hallucination and introduce both expense and instability, making them difficult to generalize to real-world application (Kalai et al., 2025; Wang et al., 2024). **Synthetic environments** reconstruct tools and databases through sandbox code generation, offering a practical compromise between realism, scalability, and training stability (Cai et al., 2025; Fang et al., 2025; Hao et al., 2026; Song et al., 2026; Wang et al., 2026). However, AutoForge (Cai et al., 2025) and AgentScaler (Fang et al., 2025) rely on pre-collected tools or documentation, EnvScaler (Song et al., 2026) builds on existing task sets, and AWM (Wang et al., 2026) starts from abstract scenario seeds, rather than directly recovering real online tool ecosystems. In contrast, ENVFACTORY autonomously discovers tools from authentic online resources, eliminating reliance on pre-curated specifications. By automatically constructing stateful databases and executable tool interfaces with rigorous verification, ENVFACTORY delivers scalable, robust environments grounded in real-world tool ecosystems.

Dependency Tool Graph. Sequential tool-use queries often involve strong dependencies among tools, making it challenging for LLMs to generate realistic trajectories directly (He et al., 2025a; Li et al., 2026a; Wu et al., 2025). A common solution constructs a directed dependency graph over available tools and samples valid sequences via graph traversal. Tool graphs are typically built using either (1) semantic similarity matching between tool parameters and descriptions (Chen et al., 2025b; Wang et al., 2025), which is efficient but may miss implicit logical relationships; or (2) LLM-based reasoning to infer dependencies (Fang et al., 2025), which is more flexible but computationally expensive and potentially inconsistent. Once constructed, these graphs are commonly traversed via naive random walks (Ma et al., 2025; Yin et al., 2025), which often fail to fully resolve dependencies—particularly when a tool requires outputs from multiple preceding tools. In contrast, our approach combines semantic matching with LLM-augmented refinement for graph construction, and introduces a topology-aware sampling strategy that recursively resolves unsatisfied input dependencies before tool selection. More related work is discussed at Appendix E.

3. Method

3.1. Problem Setup: Tool Agentic Interaction

We define the tool agentic interaction between users, agents, and environments as follow:

Environments ($\mathcal{E}$). Let $\mathcal{E}$ denote the set of available tool environments. Each environment $e \in \mathcal{E}$ is defined as $e = (m, \mathcal{D}, \pi, \mathcal{V}_e)$, where m denotes environment metadata (e.g., descriptions, tool definitions, and tool schemas), $\mathcal{D}$ is the stateful database schema specifying the underlying environment state, π is the executable Python implementation, and $\mathcal{V}_e$ is the tool interface exposed to the agent (e.g., tool names, descriptions, and parameter specifications), use MCP (Anthropic, 2024) by default.

Tools ($\mathcal{V}$). Each environment $e \in \mathcal{E}$ exposes a tool interface $\mathcal{V}_e$, and the global toolset is defined as $\mathcal{V} = \bigcup_{e \in \mathcal{E}} \mathcal{V}_e$. Each tool $v \in \mathcal{V}$ is associated with an input space $\mathcal{I}(v)$ and an output space $\mathcal{O}(v)$.

Agent. At each step, the agent observes the user message or tool execution results, and chooses either to invoke tools from $\mathcal{V}$ or to emit a natural-language response to the user.

User. When receiving the agent's message, the user may provide additional information, clarify the agent's questions, or perform instructed actions.

For each turn, the interaction continues until either a predefined maximum number of steps is reached or the user proactively terminates the conversation by emitting a stop token.

Overview. To synthesize high-quality tool agentic interaction trajectories, EnvFactory first constructs environments autonomously using *EnvGen*, yielding an executable environment set $\mathcal{E}$ and corresponding tool set $\mathcal{V}$. Using $\mathcal{V}$, we build a dependency tool graph G that captures relationships among tools. Leveraging G, we then employ a topology-aware sampling strategy to randomly sample an ordered list of tools $\tau = [v_1, ..., v_n]$, which serves as the backbone for synthesizing multi-step, multi-turn tool agentic interaction trajectories using *QueryGen*.

3.2. Environment Construction

Overview. Given an empty set of environment $\mathcal{E} = \emptyset$, *EnvGen* fully automates the construction of a new environment $e_{\text{new}} = (m, \mathcal{D}, \pi, \mathcal{V}_{e_{\text{new}}}) \notin \mathcal{E}$ by generating diverse proposals, retrieving authentic sources, and iteratively implementing, executing, and revising to ensure a stable training environment, as shown in Figure 1. The environment pool is subsequently augmented as $\mathcal{E} \leftarrow \mathcal{E} \cup \{e_{\text{new}}\}$.

Proposal and Sketch. Instead of drafting environments from static documents, our Search Agent

plans and sketches candidate environments with authentic external sources. The agent analyzes the current environments $\mathcal{E}$ to identify coverage gaps and retrieves source-grounded, broadly applicable functionalities—such as API documentation, technical reports, and usage examples—to inform environment designs. For each selected candidate, it then produces structured metadata m, including environment descriptions, tool definitions, and tool schemas, which serve as a blueprint for constructing e_{new}. By grounding environment proposals in authentic and widely applicable functionalities, this stage promotes the diversity, authenticity, and scalability of the generated environments.

Database Modeling. Given metadata m, a Code Agent derives a stateful database schema $\mathcal{D}$ that captures the entities, relationships, and mutable states needed to support the environment's functionalities. Tool parameters, intermediate states, and persistent records are formalized as Pydantic schemas with standardized serialization interfaces for loading and dumping states. This design ensures clean session isolation and reproducible execution across training rollouts.

Code Implementation. Conditioned on m and $\mathcal{D}$, the Code Agent implements executable Python code π for each tool, ensuring consistency with the specified functionality, constraints, and schema definitions. The implementations are then wrapped into a standardized tool interface $\mathcal{V}_{e_{\text{new}}}$ (e.g., MCP), exposing well-defined tool names, descriptions, and parameter specifications to agents.

Revision Loop. After constructing $\mathcal{D}$, π, and $\mathcal{V}_e$, a Test Agent creates unit test cases and validates the environment against four criteria: (1) tool interfaces are consistent with metadata m; (2) tools import and execute successfully; (3) execution results match expected behavior; and (4) database states transition correctly after tool invocation. Upon failure, the Test Agent produces a structured error report that localizes the source (e.g., implementation logic) and provides revision suggestions. The Code Agent then updates the corresponding component and rebuilds the environment. This iterative validation-and-revision loop continues until all tests pass or a maximum revision budget is reached. The final verified environment $e_{\text{new}} = (m, \mathcal{D}, \pi, \mathcal{V}_{e_{\text{new}}})$ is cross-validated across all components, ensuring stable and reproducible execution during RL training.

3.3. Dependency Tool Graph

3.3.1. Tool Graph Construction

We construct a tool dependency graph $G = (\mathcal{V}, E)$ using semantic matching to capture the nonlinear relationships between tools. However, relying solely on semantic similarity is insufficient to model all logical dependencies. For instance, tools without input or output parameters and tools that belong to the same functional group despite differing signatures may not be adequately represented. To address these limitations, we propose a fine-grained method that models both tools and their parameters as nodes in G, resulting in a graph that is more semantically coherent and logically sound.

Step 1: Semantic Parameter Matching. Using the `BAAI/bge-m3` embedding model (Chen et al., 2025a), we encode all input and output parameters of every tool. For any pair of tools $(v_i, v_j) \in \mathcal{V} \times \mathcal{V}$, we compute the cosine similarity between the embeddings of every output parameter $p_o \in \mathcal{O}(v_i)$ and every input parameter $p_i \in \mathcal{I}(v_j)$. If any such similarity exceeds a preset threshold, we add a directed edge $(v_i \to v_j)$ to G, indicating that v_j may consume outputs produced by v_i.

Step 2: Logical Dependency Refinement. For each environment $e \in \mathcal{E}$, we further prompt a LLM to analyze the tools in $\mathcal{V}_e$, identify missing logical dependencies and prune spurious edges introduced by semantic matching. This step is essential because parameter-less tools will be otherwise isolated. For example, in the `Notion` environment, the tool delete_all_notes accepts no input parameters and returns no output parameters; without further refinement, it would be disconnected from the graph.

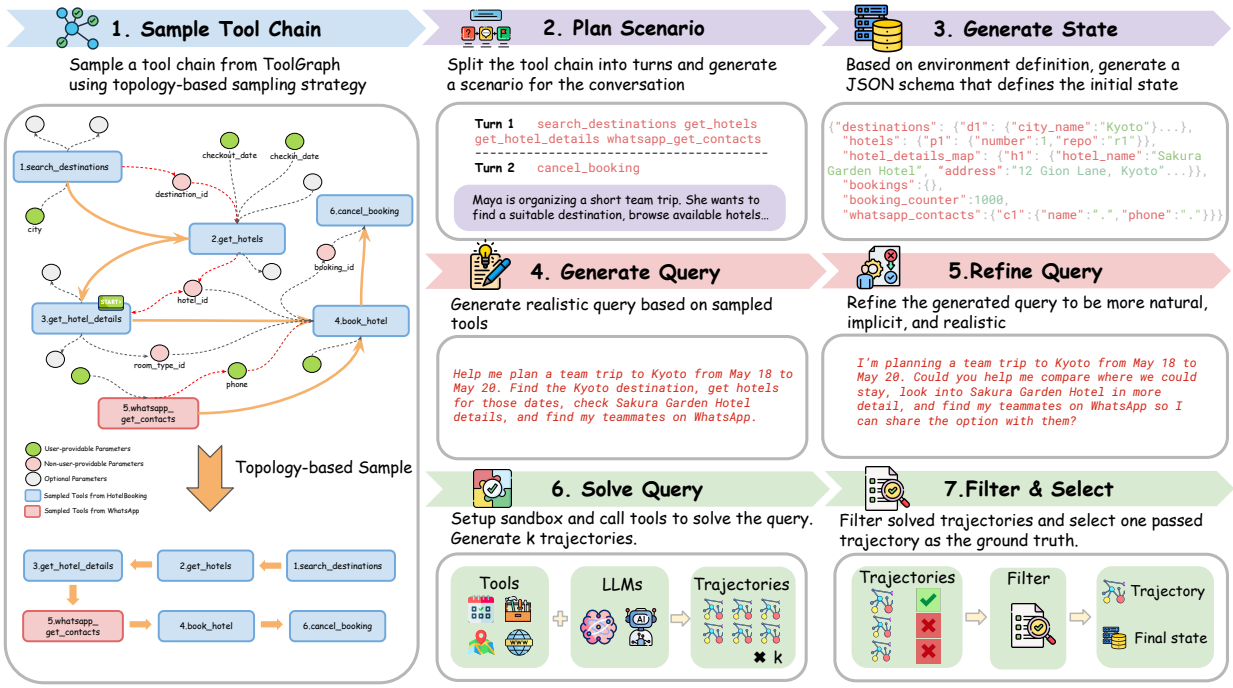

Figure 2 | The overall framework of *QueryGen*: Part 1 illustrates the topology-aware sampling strategy, highlighting its non-linear nature, while Parts 2–7 detail the step-by-step synthesis of queries.

3.3.2. Topology-Aware Sampling

Leveraging the tool graph G, we sample a tool sequence $\tau = [v_1, v_2, \ldots, v_n]$ to guide the synthesis of realistic tool-use queries. However, two challenges bottleneck this process. First, vanilla sampling strategies such as random walk only capture sequential logic, whereas real-world scenarios often demand non-linear reasoning patterns. Second, synthesizing natural user queries from sampled tool chains requires that missing input parameters be realistically satisfiable—either provided explicitly by the user or derived from the outputs of preceding tools in the chain. To address both challenges, we enforce the following sampling constraint: *All required input parameters of a sampled tool must be either externally provided by the human user or internally derived from the outputs of previously sampled tools.* Figure 2 shows an example of topology-aware sampling strategy.

Identify Internal and External Parameters. We employ an LLM to classify each input parameter as either *external* or *internal*. External parameters (e.g., `city`, `name`) require explicit provision from an external source such as a human user. In contrast, internal parameters (e.g., `hotel_id` for `book_hotel`) depend on the outputs of preceding tool calls (e.g., `get_hotel_list`), representing internal system states that users are unlikely to know or recall.

Sample Dependencies. When sampling a tool v, an input parameter $p_i \in \mathcal{I}(v)$ is deemed *independent* if it satisfies at least one of the following conditions: 1). *Optional*: p_i has a default value or can be omitted; 2). *Externally providable*: p_i is classified as *external* so it can be naturally provided by the users; 3). *Internally satisfiable*: p_i is classified as *internal* but it's also an output of previously sampled tool in τ. For any *dependent* parameter p_i, the sampler recursively selects a *prior* tool capable of generating it by traversing backward along the inverse edges of G. This recursive process ensures that all dependencies are resolved before v is added to τ. Additionally, to encourage diversity, the sampler may stochastically introduce a prior tool for a resolvable parameter with a small probability p. The full algorithmic details are provided in Appendix H.

Sample Neighbors. Once all dependencies for v are resolved, the sampler randomly selects 1 to k

neighbors (with equal probability) along the outgoing edges from v to extend the tool chain. This branching mechanism enables non-linear tool-use patterns beyond simple sequential chains, guiding more complex tool-use trajectory synthesis.

3.4. Tool-Use Trajectory Synthesis

Overview. Using a topology-aware sampling strategy, we sample tool chains τ subject to logical dependency constraints. Based on τ, *QueryGen* synthesizes multi-turn, multi-step tool-use trajectories through two principles: (1) *Realistic user intent:* iteratively generating and refining naturalistic intents to reflect real-world pragmatic patterns such as implicit reasoning and ambiguity; and (2) *Verifiable ground-truth:* deploying sandboxed agentic interaction to produce verified tool-call trajectories that ensure reliable reward signals. The prompts can be found in Appendix I.

Planning. Grounded on τ, we first construct a user profile and scenario. From this scenario, we derive a database state strictly conforming to the schema in Section 3.2. We then stochastically partition the tool chain into multiple dialogue turns, each comprising 1–5 randomly sampled tools.

Generation and Refinement. For each turn, the *QueryGen* synthesizes a naturalistic user query conditioned on the current database state, dialogue history, and sampled tools through two stages: (i) *Subgoal decomposition*, where tools are broken into fine-grained subgoals and user intents, and (ii) *Goal articulation*, where natural language requests are composed from these subgoals. Because initially generated queries often lack the implicit reasoning and conciseness characteristic of human language, the *QueryGen* enhances realism through four calibrated refinement: (1) *Implicit reference:* replacing explicit identifiers with contextual references and omitting deducible parameters; (2) *Action compression:* compressing logically inferable intermediate steps; (3) *Ambiguity introduction*: introducing reasonable referential ambiguity; and (4) *Goal expansion:* augmenting queries with plausible, thematically related secondary objectives. With decomposition and refinement, the synthesized query reflects the pragmatic and implicit nature of real user requests.

Agentic Interaction. To obtain ground-truth tool-call trajectories, we deploy sandbox environments with agents and simulated users, mirroring the RL training setup. At each turn, the agent resolves the generated queries by invoking tools, issuing explicit instructions to the user, or requesting clarification while the user follows the instructions from the agent, answers the questions, or proactively ends the conversation based on the feedback. This process continues until the user actively terminates the conversation or the maximum step limit is reached. We independently generate k candidate solution trajectories to ensure comprehensive coverage of the solution space. Simulated users details can be found in Appendix F.

Evaluation. Given k candidate trajectories and their corresponding database state changes, the pipeline evaluates each solution and selects the one that optimally resolves the query. A filtering process then removes redundant tool calls and unnecessary user interactions, and a masking process annotates the arguments whose values do not affect tool-use correctness for each retained tools.

3.5. Model Training

With the synthesized trajectories, we perform post-training using both SFT and RL. For RL, evaluating tool-use correctness is non-trivial because valid executions are often non-unique and cannot be determined solely from reference trajectories or final database states. For example, independent read-only tool calls may be invoked in different orders, and parameters such as `limit` may vary across equally valid executions. To account for this ambiguity, we use a composite reward with three components: 1) *trajectory-based reward* that measures the matches between the predicted

and ground-truth tool-calling sequences; 2) *state-based reward* that evaluates the equivalence of the final database states after tool execution; and 3) *length penalty* that discourages unnecessarily long tool-call sequences. The overall reward is:

$$R = \alpha \cdot R_{\text{traj}} + (1 - \alpha) \cdot R_{\text{state}} - \gamma \cdot P_{\text{length}}$$

where $R_{\text{traj}} \in [0, 1]$ is the trajectory-based reward, R_{state} is the state-based reward, P_{length} is the length penalty and $\alpha, \gamma \geq 0$ are the weighting coefficients.

4. Experiments and Analysis

4.1. Setup

Data Statistics. We construct 85 diverse MCP environments spanning seven domains: commerce, finance, travel, office, lifestyle, research, and utilities. From these environments, we synthesize 1,622 conversations for SFT trajectories and 953 conversations for RL trajectories. On average, each conversation comprises 4.82 turns, with each turn containing 3.29 steps—including both tool calls and user interactions. Further details are provided in Figure 5.

Baselines and Benchmarks. We adopt Qwen3-(1.7B, 4B, 8B) (Yang et al., 2025a) as training backbones. For baseline comparison, we directly use available checkpoints from AWM (Wang et al., 2026) and EnvScaler (Song et al., 2026), two concurrent work on tool-use trajectory synthesis. Evaluation is conducted on BFCL v3 (Patil et al., 2025), τ^2-Bench (Barres et al., 2025), VitaBench (He et al., 2025b), and MCP-Atlas (Bandi et al., 2026). Further details are provided in Appendix F.

Implementation Details. Our training pipeline consists of: *Stage 1*: SFT initialized with user interaction trajectories; *Stage 2*: RL training uses only tool-call trajectories. We perform SFT using LlamaFactory (Zheng et al., 2024b) and RL using VeRL (Sheng et al., 2024) with GRPO (Shao et al., 2024b). Details are provided in Appendix F.

4.2. Main Results

Table 1 presents a comprehensive comparison across four benchmarks and strong baselines.

SFT Cold Start Delivers the Largest Relative Gains. Supervised fine-tuning on our automatically generated trajectories alone yields substantial improvements across diverse tool-use benchmarks. On BFCL multi-turn evaluation, EnvFactory (SFT) improves Qwen3-1.7B from 16.75 to 23.25 and Qwen3-4B from 33.50 to 44.25. Similar gains are observed on τ^2-Bench, where Qwen3-1.7B improves from 14.61 to 15.57, while Qwen3-4B achieves a strong gain on the challenging retail domain (38.60 $\rightarrow$ 47.37). The improvements further generalize to more challenging benchmarks. On MCP-Atlas, pass rates nearly double across all model scales, e.g., from 4.12 to 7.90 for Qwen3-4B and from 5.15 to 8.25 for Qwen3-8B. On VitaBench, Qwen3-1.7B improves from 1.33 to 6.33, while Qwen3-4B improves from 7.67 to 11.33. 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. Building on the strong SFT initialization, RL training consistently yields further gains across nearly all benchmarks and model scales. Compared with EnvFactory (SFT), the full EnvFactory improves the overall score from 18.60 to 19.74 for Qwen3-1.7B, from 27.29 to 30.77 for Qwen3-4B, and from 30.82 to 33.40 for Qwen3-8B. The improvements are particularly evident on challenging interactive benchmarks. On VitaBench, Qwen3-4B improves from 11.33 to 16.00, while on MCP-Atlas, Qwen3-8B substantially improves pass rate from 8.25 to 13.75 and mean coverage from 22.86 to 25.98. Similar gains are observed on BFCL

Table 1 | Experiment results on BFCL, τ^2-Bench, VitaBench, and MCP-Atlas. **Cell** and Cell indicate the best and second-best results for each evaluation metric, respectively, while **Cell** and Cell denote methods that achieve stronger performance with fewer environments and training tasks.

	Data Scale		BFCL		MCP-Atlas		τ^2-Bench				VitaBench				Overall
Model	*Env.*	*Tasks*	*Single Turn*	*Multi Turn*	*Pass Rate*	*Mean Cov.*	*Airline*	*Retail*	*Tele*	**Avg.**	*Deliver*	*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
Our (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
Our	**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
Our (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
Our	**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
Our (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
Our	**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**

multi-turn evaluation, where Qwen3-4B improves from 44.25 to 48.50 and Qwen3-8B from 46.50 to 49.00. These results suggest that SFT provides strong foundational tool-use behaviors and RL further enhances reasoning and execution robustness.

Strong Generalization Across Benchmark Types. ENVFACTORY demonstrates consistent improvements across both conversational benchmarks (τ^2-Bench and VitaBench) and non-conversational benchmarks (BFCL and MCP-Atlas). On conversational benchmarks, Qwen3-4B improves from 25.25 to 30.13 on τ^2-Bench and from 7.67 to 16.00 on VitaBench, while Qwen3-8B achieves the best conversational performance with 33.67 and 18.67, respectively. At the same time, ENVFACTORY substantially improves non-conversational tool-use capability, boosting BFCL multi-turn accuracy from 33.50 to 48.50 for Qwen3-4B and achieving the best MCP-Atlas results with a 13.75 pass rate and 25.98 mean coverage on Qwen3-8B. These results demonstrate that ENVFACTORY generalizes effectively across both conversational interaction and compositional tool-execution settings.

4.3. Effect of the Environments Scaling

Figure 3 studies how the number of executable environments affects tool-use learning in ENVFACTORY. To evaluate scaling behavior, we construct two additional training subsets with 50 and 75 randomly sampled environments, respectively, and perform the same SFT+RL training procedure on each subset. As shown in Figure 3(a), increasing the environment pool consistently improves BFCL-v3 multi-turn performance across Qwen3-1.7B, Qwen3-4B, and Qwen3-8B. This trend indicates that broader environment coverage exposes the model to more diverse tool schemas, state transitions, and multi-step interaction patterns, improving generalization to unseen tool-use tasks. The scaling curve also shows a diminishing-return pattern: the gain from 50 to 75 environments is larger than that

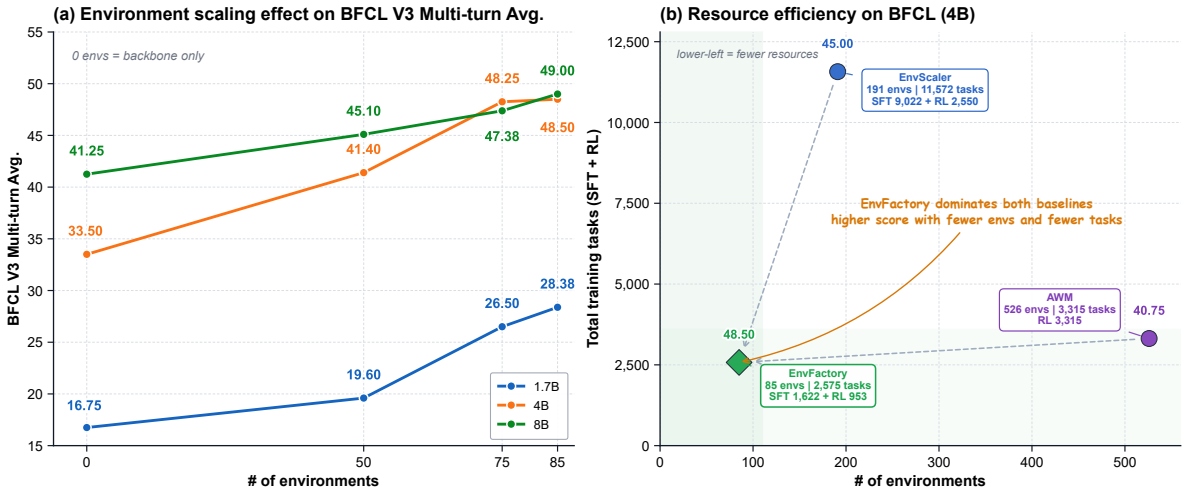

Figure 3 | Environment scaling and resource efficiency analysis on BFCL-v3. (a) BFCL-v3 multi-turn average performance under different numbers of environments across Qwen3-1.7B, Qwen3-4B, and Qwen3-8B. (b) Resource efficiency comparison on Qwen3-4B, where the x-axis denotes the number of environments, the y-axis denotes the total number of training tasks, and the marker label reports the BFCL-v3 multi-turn average score.

from 75 to 85 environments, suggesting that later additions may contain more overlapping tool logic or task structures. Figure 3(b) further shows that ENVFACTORY achieves stronger BFCL-v3 multi-turn performance while using only 85 environments and 2,575 training tasks, far fewer than the baselines. This result suggests that verified stateful environments and dependency-aware trajectories provide effective supervision and reward signals from a compact training set.

4.4. Ablation Study

Table 2 | Performance of training with direct RL.

Model	BFCL Single-turn	BFCL Multi-turn	τ^2 Bench	VitaBench
Qwen3-1.7B	79.48	16.75	14.67	1.33
Our-1.7B (RL)	**79.53**	**18.33**	**18.28**	**1.67**
Qwen3-4B	85.15	33.50	**25.33**	7.67
Our-4B (RL)	**85.26**	**41.38**	24.83	**12.74**
Qwen3-8B	84.31	41.25	**32.33**	16.70
Our-8B (RL)	**84.42**	**44.35**	29.08	**17.00**

Table 3 | Performance comparison between refined and unrefined trajectories for SFT.

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

Experiment Results on Direct RL.

We examine whether ENVFACTORY-generated trajectories can directly support RL training without an SFT cold-start phase. As shown in Table 2, direct RL improves several interactive benchmarks, such as BFCL multi-turn accuracy for ENVFACTORY-4B (33.50 to 41.38) and τ^2-Bench for ENVFACTORY-1.7B (14.67 to 18.28). However, these gains are smaller and less stable than RL after SFT, indicating that SFT initialization remains important for stable policy optimization.

Effects of the Refinement Stage. To study the impact of the refinement stage in query generation,

we synthesize 250 SFT trajectories with and without refinement, respectively. Table 3 shows that refined trajectories consistently outperform unrefined ones, especially on ambiguous settings such as `Miss-Func` and `Miss-Param`. This suggests that refinement improves query ambiguity calibration and provides higher-quality supervision.

Effects of the Reward Weighting Coefficient. We conduct an ablation over the trajectory-based reward weighting coefficient $\alpha \in \{0, 0.3, 0.5, 0.7, 1.0\}$ on BFCL while fixing the length penalty coefficient γ. Figure 4 shows that relying only on state-based reward ($\alpha = 0$) or only on trajectory matching ($\alpha = 1.0$) degrades performance. Balanced weighting performs better, with $\alpha = 0.5$ achieving the best peak accuracy of 41.38%. Removing either reward component altogether hurts performance, indicating that both trajectory fidelity and state equivalence are necessary for effective RL training.

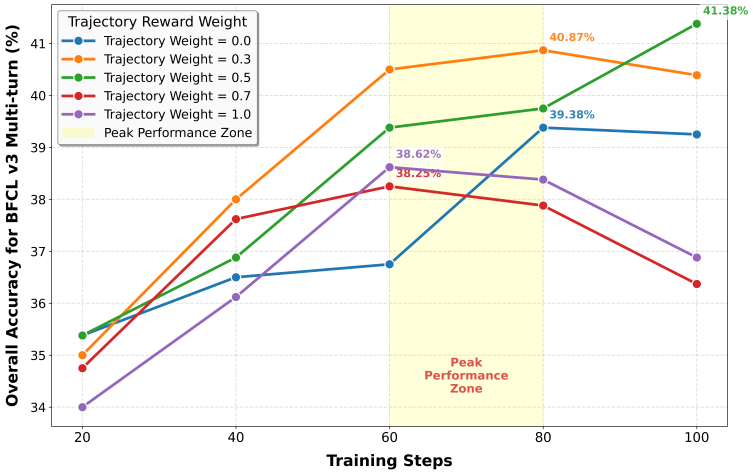

Figure 4 | Ablation results on BFCL-v3 under different trajectory reward weights.

5. Conclusion

We presented EnvFactory, a fully automated framework that addresses two critical bottlenecks in Agentic RL for tool-use: the lack of scalable, verifiable environments and the scarcity of realistic, implicitly-reasoned training trajectories. Unlike prior approaches that rely on costly production APIs, hallucination-prone simulators, or static synthetic environments, EnvFactory autonomously constructs verified, stateful environments by exploring real-world online resources and recursively resolving logical dependencies among tools. It further bridges the realism gap by transforming over-specified instruction lists into natural human-like requests through calibrated refinement that injects implicit intents and ambiguity. Experimental results show that EnvFactory consistently outperforms strong baselines in both training efficiency and downstream performance, while requiring significantly fewer synthetic environments and samples.

References

Anthropic. Model context protocol. https://www.anthropic.com/news/model-context-protocol, Nov. 2024. Accessed: 2026-05-05.

C. Bandi, B. Hertzberg, G. Boo, T. Polakam, J. Da, S. Hassaan, M. Sharma, A. Park, E. Hernandez, D. Rambado, I. Salazar, R. Cruz, C. Rane, B. Levin, B. Kenstler, and B. Liu. Mcp-atlas: A large-scale benchmark for tool-use competency with real mcp servers, 2026. URL https://arxiv.org/abs/2602.00933.

V. Barres, H. Dong, S. Ray, X. Si, and K. Narasimhan. τ^2-bench: Evaluating conversational agents in a dual-control environment, 2025. URL https://arxiv.org/abs/2506.07982.

S. Cai, R. Fang, J. Wu, B. Li, X. Wang, Y. Jiang, L. Su, L. Zhang, W. Yin, Z. Zhang, F. Feng, P. Xie, and X. Wang. Autoforge: Automated environment synthesis for agentic reinforcement learning, 2025. URL https://arxiv.org/abs/2512.22857.

J. Chen, S. Xiao, P. Zhang, K. Luo, D. Lian, and Z. Liu. M3-embedding: Multi-linguality, multi-functionality, multi-granularity text embeddings through self-knowledge distillation, 2025a. URL https://arxiv.org/abs/2402.03216.

W. Chen, W. Li, D. Yao, X. Meng, C. Gong, and J. Bi. Gtool: Graph enhanced tool planning with large language model, 2025b. URL https://arxiv.org/abs/2508.12725.

Z. Chen, Z. Zhao, K. Zhang, B. Liu, Q. Qi, Y. Wu, T. Kalluri, S. Cao, Y. Xiong, H. Tong, H. Yao, H. Li, J. Zhu, X. Li, D. Song, B. Li, J. Weston, and D. Huynh. Scaling agent learning via experience synthesis, 2025c. URL https://arxiv.org/abs/2511.03773.

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

R. Fang, S. Cai, B. Li, J. Wu, G. Li, W. Yin, X. Wang, X. Wang, L. Su, Z. Zhang, S. Wu, Z. Tao, Y. Jiang, P. Xie, F. Huang, and J. Zhou. Towards general agentic intelligence via environment scaling, 2025. URL https://arxiv.org/abs/2509.13311.

J. Feng, S. Huang, X. Qu, G. Zhang, Y. Qin, B. Zhong, C. Jiang, J. Chi, and W. Zhong. Retool: Reinforcement learning for strategic tool use in llms, 2025. URL https://arxiv.org/abs/2504.11536.

Z. Gao, L. Chen, J. Zhou, and B. Dai. One-shot entropy minimization. *arXiv preprint arXiv:2505.20282*, 2025.

Z. Guo, S. Cheng, H. Wang, S. Liang, Y. Qin, P. Li, Z. Liu, M. Sun, and Y. Liu. Stabletoolbench: Towards stable large-scale benchmarking on tool learning of large language models, 2025. URL https://arxiv.org/abs/2403.07714.

B. Hao, Z. Xu, M. Wang, Y. Wen, Y. Chen, C. Peng, L. Chen, D. Wang, X. Zhao, J. Gu, C. Zhuang, and J. Zhang. Reasoning through exploration: A reinforcement learning framework for robust function calling, 2025. URL https://arxiv.org/abs/2508.05118.

B. Hao, Z. Xu, Y. Wen, X. Xu, Y. Liu, T. Zhao, M. Wang, L. Chen, D. Wang, Y. Chen, C. Peng, X. Zhao, C. Zhuang, and J. Zhang. From failure to mastery: Generating hard samples for tool-use agents, 2026. URL https://arxiv.org/abs/2601.01498.

P. He, Z. Dai, B. He, H. Liu, X. Tang, H. Lu, J. Li, J. Ding, S. Mukherjee, S. Wang, Y. Xing, J. Tang, and B. Dumoulin. Traject-bench:a trajectory-aware benchmark for evaluating agentic tool use, 2025a. URL https://arxiv.org/abs/2510.04550.

W. He, Y. Sun, H. Hao, X. Hao, Z. Xia, Q. Gu, C. Han, D. Zhao, H. Su, K. Zhang, M. Gao, X. Su, X. Cai, X. Cai, Y. Yang, and Y. Zhao. Vitabench: Benchmarking llm agents with versatile interactive tasks in real-world applications, 2025b. URL https://arxiv.org/abs/2509.26490.

B. Jin, H. Zeng, Z. Yue, J. Yoon, S. Arik, D. Wang, H. Zamani, and J. Han. Search-r1: Training llms to reason and leverage search engines with reinforcement learning, 2025. URL https://arxiv.org/abs/2503.09516.

A. T. Kalai, O. Nachum, S. S. Vempala, and E. 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.

S. Li, Y. Huang, Z. Liu, Z. Li, J. fu, L. Song, J. Bian, J. Zhang, and R. Wang. Experience-evolving multi-turn tool-use agent with hybrid episodic-procedural memory, 2026a. URL https://arxiv.org/abs/2512.07287.

Y. Li, H. A. Inan, X. Yue, W.-N. Chen, L. Wutschitz, J. Kulkarni, R. Poovendran, R. Sim, and S. Rajmohan. Simulating environments with reasoning models for agent training, 2025. URL https://arxiv.org/abs/2511.01824.

Y. Li, H. Wang, J. Qiu, Z. Yin, D. Zhang, C. Qian, Z. Li, P. Ma, G. Chen, and H. Ji. From word to world: Can large language models be implicit text-based world models?, 2026b. URL https://arxiv.org/abs/2512.18832.

X. Liang, Z. Li, Y. Gong, Y. Shen, Y. N. Wu, Z. Guo, and W. Chen. Beyond pass@ 1: Self-play with variational problem synthesis sustains rlvr. *arXiv preprint arXiv:2508.14029*, 2025.

J. Luo, W. Zhang, Y. Yuan, Y. Zhao, J. Yang, Y. Gu, B. Wu, B. Chen, Z. Qiao, Q. Long, R. Tu, X. Luo, W. Ju, Z. Xiao, Y. Wang, M. Xiao, C. Liu, J. Yuan, S. Zhang, Y. Jin, F. Zhang, X. Wu, H. Zhao, D. Tao, P. S. Yu, and M. Zhang. Large language model agent: A survey on methodology, applications and challenges, 2025. URL https://arxiv.org/abs/2503.21460.

S. Ma, X. Jiang, C. Xu, C. Yang, L. Zhang, and J. Guo. Synthesize-on-graph: Knowledgeable synthetic data generation for continue pre-training of large language models, 2025. URL https://arxiv.org/abs/2505.00979.

L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

S. G. Patil, H. Mao, F. Yan, C. C. Ji, V. Suresh, I. Stoica, and J. E. Gonzalez. The berkeley function calling leaderboard (BFCL): from tool use to agentic evaluation of large language models. In A. Singh, M. Fazel, D. Hsu, S. Lacoste-Julien, F. Berkenkamp, T. Maharaj, K. Wagstaff, and J. Zhu, editors, *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.

Y. Qin, S. Liang, Y. Ye, K. Zhu, L. Yan, Y. Lu, Y. Lin, X. Cong, X. Tang, B. Qian, et al. Toolllm: Facilitating large language models to master 16000+ real-world apis. In *The twelfth international conference on learning representations*, 2023.

C. Qu, S. Dai, X. Wei, H. Cai, S. Wang, D. Yin, J. Xu, and J.-r. Wen. Tool learning with large language models: a survey. *Frontiers of Computer Science*, 19(8), Jan. 2025. ISSN 2095-2236. doi: 10.1007/s11704-024-40678-2. URL http://dx.doi.org/10.1007/s11704-024-40678-2.

R. Rafailov, A. Sharma, E. Mitchell, C. D. Manning, S. Ermon, and C. Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in neural information processing systems*, 36:53728–53741, 2023.

Z. Shao, Z. Liu, W. Zhang, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024a.

Z. Shao, P. Wang, Q. Zhu, R. Xu, J. Song, X. Bi, H. Zhang, M. Zhang, Y. K. Li, Y. Wu, and D. Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024b. URL https://arxiv.org/abs/2402.03300.

G. Sheng, C. Zhang, Z. Ye, X. Wu, W. Zhang, R. Zhang, Y. Peng, H. Lin, and C. Wu. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv: 2409.19256*, 2024.

X. Song, H. Chang, G. Dong, Y. Zhu, J.-R. Wen, and Z. Dou. Envscaler: Scaling tool-interactive environments for llm agent via programmatic synthesis, 2026. URL https://arxiv.org/abs/2601.05808.

M. Sullivan, M. Hartmann, and A. Koller. Procedural environment generation for tool-use agents, 2025. URL https://arxiv.org/abs/2506.11045.

R. Wang, G. Todd, Z. Xiao, X. Yuan, M.-A. Côté, P. Clark, and P. Jansen. Can language models serve as text-based world simulators?, 2024. URL https://arxiv.org/abs/2406.06485.

Z. Wang, X. Zeng, W. Liu, L. Li, Y. Wang, L. Shang, X. Jiang, Q. Liu, and K.-F. Wong. Toolflow: Boosting llm tool-calling through natural and coherent dialogue synthesis, 2025. URL https://arxiv.org/abs/2410.18447.

Z. Wang, C. Xu, B. Liu, Y. Wang, S. Han, Z. Yao, H. Yao, and Y. He. Agent world model: Infinity synthetic environments for agentic reinforcement learning, 2026. URL https://arxiv.org/abs/2602.10090.

J. Wu, Q. Zhao, Z. Chen, K. Qin, Y. Zhao, X. Wang, and Y. Yao. Gap: Graph-based agent planning with parallel tool use and reinforcement learning, 2025. URL https://arxiv.org/abs/2510.25320.

Z. Xu, A. M. Soria, S. Tan, A. Roy, A. S. Agrawal, R. Poovendran, and R. Panda. Toucan: Synthesizing 1.5m tool-agentic data from real-world mcp environments, 2025. URL https://arxiv.org/abs/2510.01179.

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

C. Yang, R. Le, Y. Xing, Z. An, Z. Chen, W. X. Zhao, Y. Song, and T. Zhang. Toolmind technical report: A large-scale, reasoning-enhanced tool-use dataset, 2025b. URL https://arxiv.org/abs/2511.15718.

Z. Yang, Z. Guo, Y. Huang, X. Liang, Y. Wang, and J. Tang. Treerpo: Tree relative policy optimization. *arXiv preprint arXiv:2506.05183*, 2025c.

Z. Yang, Z. Guo, Y. Huang, Y. Wang, D. Xie, H. Li, Y. Wang, X. Liang, and J. Tang. Depth-breadth synergy in rlvr: Unlocking llm reasoning gains with adaptive exploration. *arXiv preprint arXiv:2508.13755*, 2025d.

J. Ye, C. Jiang, Z. Du, Y. Xu, X. Yao, Z. Xi, X. Fan, Q. Zhang, T. Gui, X. Huang, and J. Chen. Feedback-driven tool-use improvements in large language models via automated build environments, 2026. URL https://arxiv.org/abs/2508.08791.

F. Yin, Z. Wang, I.-H. Hsu, J. Yan, K. Jiang, Y. Chen, J. Gu, L. T. Le, K.-W. Chang, C.-Y. Lee, H. Palangi, and T. Pfister. Magnet: Multi-turn tool-use data synthesis and distillation via graph translation, 2025. URL https://arxiv.org/abs/2503.07826.

Y. Yue, Z. Chen, R. Lu, A. Zhao, Z. Wang, S. Song, and G. Huang. Does reinforcement learning really incentivize reasoning capacity in llms beyond the base model? *arXiv preprint arXiv:2504.13837*, 2025.

L. Zheng, L. Yin, Z. Xie, C. Sun, J. Huang, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024a.

Y. Zheng, R. Zhang, J. Zhang, Y. Ye, and Z. Luo. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 3: system demonstrations)*, pages 400–410, 2024b.

X. Zhu, M. Xia, Z. Wei, W.-L. Chen, D. Chen, and Y. Meng. The surprising effectiveness of negative reinforcement in llm reasoning. *arXiv preprint arXiv:2506.01347*, 2025.

A. Broader Impact

This work introduces a framework for the automated construction of executable environments and realistic trajectories, significantly lowering the barrier for developing robust AI agents capable of complex tool-use. By providing a scalable alternative to costly production APIs and hallucination-prone simulations, our approach facilitates the democratization of Agentic RL research, enabling a broader range of researchers to train agents in diverse, high-fidelity domains such as finance, research, and office automation. Furthermore, by injecting realistic human communication patterns—such as implicit intents and ambiguity—into synthetic data, this research moves AI agents closer to safe and effective real-world deployment, ensuring they can better interpret and act upon human needs.

However, the automation of agent training and environment synthesis carries potential risks that necessitate responsible oversight. The ability to rapidly generate executable tool-use ecosystems could be misused to simulate and automate malicious activities, such as large-scale fraudulent financial transactions or sophisticated phishing campaigns, if applied to sensitive domains without safeguards. Additionally, since the framework relies on online resources and LLM-guided proposals, it may inadvertently encode or amplify biases present in its source data or underlying models. To mitigate these risks, we have documented our dataset and environment construction process transparently, released our artifacts under restrictive licenses to prevent misuse, and encourage the integration of rigorous safety constraints within the synthesized environments to ensure that agents remain aligned with ethical and legal standards.

B. Limitations

EnvFactory uses the MCP as its tool interface. The MCP servers we design are stateful: write-capable tools can modify a shared environment database, which forces strict session isolation to prevent cross-contamination. As a result, each conversation requires a dedicated transport connection to the target servers, constraining the degree of parallel tool invocation and creating a throughput bottleneck during large-scale data synthesis. We mitigate this limitation by implementing an asynchronous synthesis pipeline that executes many isolated sessions concurrently, thereby maximizing overall generation efficiency despite the per-connection requirement.

C. LLM Usage Declaration

This manuscript uses LLMs strictly for the purpose of language editing and textual polishing to enhance presentation quality. We declare that the novel ideas, methodological framework, experimental execution, and data analysis are the original work of the authors. All content modified by AI tools has been carefully reviewed and validated by the authors to ensure accuracy.

D. Compute Usage

GPU Usage. We report the GPU resources required for each stage of our pipeline. For SFT data synthesis, we deploy `Qwen3-30B-A3B-Thinking-2507` on 2×80GB GPUs to generate data and distill reasoning processes. Synthesizing 1,000 multi-turn, multi-step trajectories requires approximately 20 GPU hours.

For SFT training, we fine-tune `Qwen3-4B` for 3 epochs using LlamaFactory (Zheng et al., 2024b) on 8×80GB GPUs, which consumes around 10 GPU hours.

For RL training, we train `Qwen3-4B` for 10 epochs using VeRL (Sheng et al., 2024) on 8 × 80GB GPUs, requiring approximately 20 GPU hours.

Token Usage. EnvFactory can autonomously scale up environments and data generation. The table below summarizes our token consumption across different stages. We note that trajectory synthesis supports asynchronous generation, enabling efficient scaling: synthesizing 1,000 multi-turn, multi-step trajectories takes roughly 20 hours (approximately 1.2 minutes per conversation).

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

Table 4 | Token consumption across environment construction and query synthesis.

E. 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 significantly pushed the boundaries of downstream performance in mathematics, coding, and agentic tasks. A prominent example is GRPO (Shao et al., 2024a), which optimizes LLMs at the group level by aggregating multiple outputs to provide diverse preference signals, thereby improving generalization. To achieve more fine-grained optimization, TreeRPO (Yang et al., 2025c) extends GRPO by replacing sparse, trajectory-level rewards with tree-sampled, step-level dense rewards to better guide intermediate reasoning steps.

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 tightly coupled with an LLM's reasoning upper bound. This limitation is often attributed to a rapid decline in model output entropy during the early stages of RLVR training, which stifles sustained exploration later on (Gao et al., 2025; 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, successfully stabilizing entropy and significantly boosting Pass@k performance. Alternatively, DARS (Yang et al., 2025d) addresses these training biases through difficulty-adaptive rollout sampling combined with large-batch training, ultimately delivering robust improvements in both Pass@1 and Pass@k reasoning performance.

F. Implementation Details

Data Synthesis Setup. In the **EnvGen** pipeline of EnvFactory, we primarily leverage Kimi-K2-Thinking (Kimi Team et al., 2025) to propose, draft, construct, and verify MCP environments. For the **QueryGen** pipeline, we employ DeepSeek-V3.2-Chat (DeepSeek-AI et al., 2025) for RL tool-use trajectories generation, while utilizing Qwen3-30B-A3B-Thinking-2507 (Yang et al., 2025a) SFT tool-use trajectories synthesis to distill the thinking process for SFT.

Reinforcement Learning Setup. We employ Group Relative Policy Optimization (GRPO) (Shao et al., 2024b) implemented with the Verl framework (Sheng et al., 2024). 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. For RL training, each interaction turn is treated as an individual training sample.

Supervised Fine-Tuning Setup. 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 the first SFT epoch. During SFT data construction, each tool-call or user-interaction step is treated as a separate training sample together with its associated reasoning trace. Failed tool calls are filtered out from the training data.

Evaluation Setup. During inference, we leverage the SGLang framework (Zheng et al., 2024a). We set the sampling temperature to 0 for non-thinking models and 0.7 for thinking models, with tensor parallelism (TP) set to 2 by default. For the user and evaluator agents in τ^2-Bench and VitaBench, we employ DeepSeek-V3.2-Chat (DeepSeek-AI et al., 2025).

MCP-Atlas Setup. Due to network connectivity constraints, our evaluation of MCP-Atlas uses a subset comprising 30 of 36 servers and 291 of 500 tasks. The following servers are excluded: `mongodb`, `oxylabs`, `brave-search`, `wikipedia`, `slack`, and `google-workspace`.

Simulated User Details. To instantiate a faithful simulation of tool-use scenarios, we first classify available MCP tools into *user tools* and *assistant tools* via LLM-based categorization. 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 then construct a simulated user by conditioning an LLM on three contextual inputs: (a) the narrative scenario, (b) the dialogue history, and (c) the current database state. To ensure realistic behavior, we constrain the user's knowledge to *external parameters* identified in Section 3.3.2—information that human users can realistically provide (e.g., personal preferences, location, time constraints). This prevents the simulated user from accessing *internal parameters* (e.g., system-generated IDs, backend state) that would be unavailable to actual users, thereby avoiding implausible responses such as verbatim recitation of complex internal identifiers.

G. Data Statistic

Table 5 | Comparison of environments and training samples between baselines with ✅ indicates higher efficiency.

Pipeline	Environments #	SFT Tasks #	RL Tasks #
AWM (Wang et al., 2026)	526	-	3315
EnvScaler (Song et al., 2026)	191	9022	2550
EnvFactory	85 ✅	1622 ✅	953 ✅

H. Algorithms

Our topology-aware sampling strategy ensures execution feasibility by guaranteeing all required inputs $\mathcal{I}(v)$ of each sampled tool v are satisfied before inclusion—addressing a key limitation of naive random walks. Operating on the directed dependency graph $G = (V, E)$ (Section 3.3.2), the algorithm proceeds in two phases for each node v:

Backward dependency resolution. Before adding v to the visited set $\hat{V}$, the algorithm recursively

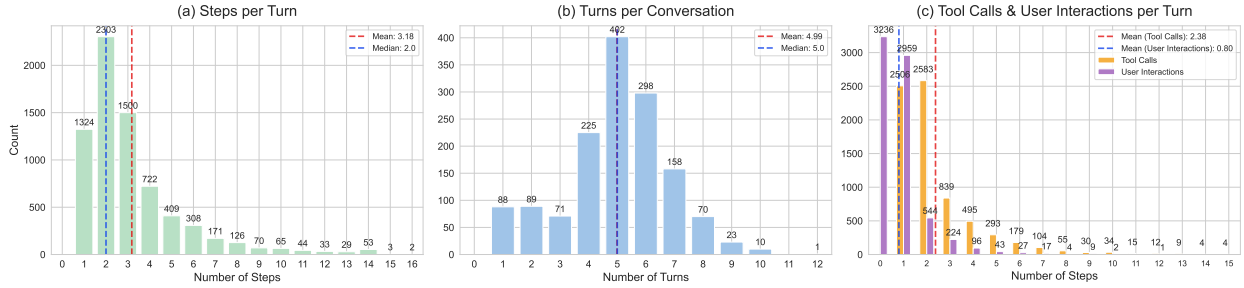

Figure 5 | Distribution of conversation statistics. (a) Number of total steps per turn. (b) Number of turns per conversation. (c) Number of tool calls steps and user interactions per turn respectively.

resolves unsatisfied inputs via `SAMPLEPRIORS`. A parameter $p_i \in \mathcal{I}(v)$ is valid if: (1) optional (has schema default), (2) user-providable per LLM classification, or (3) already produced by some $u \in \hat{V}$ where $p_i \in O(u)$. Invalid parameters trigger backward traversal to uniformly sample a producer tool u satisfying $(u \to v) \in E$ and $p_i \in O(u)$, with recursion depth capped at $D_{\max} = 3$. A stochastic override ($p = 0.1$) occasionally introduces additional priors for valid parameters to enhance trajectory diversity.

Forward expansion. Once all dependencies are resolved and incorporated into $\hat{V}$, v is added to $\hat{V}$ and the algorithm samples one outgoing neighbor from $N(v) = \{u \mid (v \to u) \in E\}$ for subsequent processing.

Algorithm 1 Topology-based Sampling Strategy

Require: $G = (V, E)$ with $|V| = N$, integer $n \leq N$, and start node v_s
Ensure: Sampled nodes $\hat{V} \subseteq V$ with $|\hat{V}| = n$

// *Initialize visited nodes set and queue for BFS*
$\hat{V} \leftarrow \{v_s\}$ and queue $\leftarrow$ Queue(v_s)
while $|\hat{V}| < n$ **do**
  $v \leftarrow$ queue.dequeue()
  // *Sample priors for current node v*
  $P(v) \leftarrow \textsc{SamplePriors}(G, \hat{V}, v, 0, D_{max})$
  for $p \in P(v)$ **do**
    if $p \notin \hat{V}$ **then**
      $\hat{V} \leftarrow \hat{V} \cup \{p\}$
    end if
  end for
  $\hat{V} \leftarrow \hat{V} \cup \{v\}$
  // *Find all neighbors of v*
  $N(v) \leftarrow \{u \in V \mid e_{vu} = 1, \forall e_{uv} \in E\}$
  // *Randomly sample a neighbor of v*
  $u \leftarrow \text{Uniform}(N(v))$
  queue.enqueue(u)
end while
return $\hat{V}$

Algorithm 2 Sample Priors

Require: Graph $G = (\mathcal{V}, \mathcal{E})$, visited nodes $\hat{V}$, current node v, current depth d, max depth $D_{\max}$
Ensure: Set of sampled prior nodes P_v

$P_v \leftarrow \emptyset$
if $d \geq D_{\max}$ **then**
  return P_v
end if
for each input parameter $p_i \in \mathcal{I}(v)$ **do**
  // Skip if p_i is valid unless stochastically overridden
  if IsValid$(p_i, \hat{V})$ and $\mathcal{U}(0,1) < 0.1$ **then**
    continue
  end if
  // Find tools that output p_i
  $C \leftarrow \{u \in \mathcal{V} \mid p_i \in \mathcal{O}(u)$ and $(u \rightarrow v) \in \mathcal{E}\}$
  if $C = \emptyset$ **then**
    continue
  end if
  // Randomly select one prior tool
  $u \leftarrow$ Uniform(C)
  if $u \notin \hat{V}$ **and** $d < D_{\max}$ **then**
    $P_u \leftarrow$ SamplePriors$(G, \hat{V}, u, d+1, D_{\max})$
    $\hat{V} \leftarrow \hat{V} \cup \{u\}$
    $P_v \leftarrow P_v \cup \{u\} \cup P_u$
  end if
end for
return P_v

I. Prompts

I.1. Prompts for 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

I.2. Prompts for 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.

I.3. Prompts for 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).