ENVFOUNDRY: A SCALABLE FRAMEWORK FOR TOOL ENVIRONMENT SYNTHESIS WITH CONTROLLABLE DEPENDENCY STRUCTURES

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) are increasingly deployed as agents that solve real-world tasks through tool use, making high-quality interactive environments essential for training and evaluation. Synthesizing these environments involves coordinating three components—task queries, toolsets, and simulators—that are coupled by *cyclic constraints*: together, they must make tasks solvable while preserving the dependencies that require multi-round interaction. Existing sequential synthesis pipelines often leave these dependencies implicit, requiring later components to accommodate already-fixed artifacts and risking infeasible tasks or unintended shortcuts. We observe that these constraints arise in part from shared relations among entities and their attributes. Based on this insight, we propose EnvFoundry, a scalable coarse-to-fine framework that makes this structure explicit through a *meta-dependency graph* (MDG). The MDG provides coordinated instructions for synthesizing all three components, aligning task requirements, tool capabilities, and environment states while controlling when information becomes accessible. This design supports consistent environment synthesis, controllable task dependencies, and extension to richer tools and state dynamics. Experiments on τ-bench, τ^2-Bench, and VitaBench show that the synthesized environments support both supervised fine-tuning and reinforcement learning, substantially improving agentic tool-use performance.

1 INTRODUCTION

Large language models (LLMs) increasingly act as agents that solve real-world tasks through tool use, including web search (Li et al., 2025a; Qiao et al., 2025), flight rescheduling (Yao et al., 2024; Qian et al., 2025), and document management (Guo et al., 2024; Qin et al., 2023). Training these agents requires high-quality interaction data, whose scarcity has motivated large-scale synthesis (DeepSeek-AI et al., 2025; Kimi Team et al., 2025; Fang et al., 2025; Prabhakar et al., 2025). Early approaches generated queries paired with tool-call annotations (Liu et al., 2024b; Patil et al., 2023), while subsequent work constructed interaction trajectories using real APIs or simulated responses (Qin et al., 2023; Guo et al., 2024; 2025). More challenging agentic tasks, however, require multiple rounds of interaction, with later actions depending on information or state changes produced by earlier calls (Wang et al., 2024; Fang et al., 2025). Synthesizing such tasks therefore requires coherent environments that sustain these dependencies throughout execution.

An agentic environment comprises three core components: **task queries**, a **toolset**, and a **simulator**. These components must jointly support feasible execution while preserving the intended dependencies between actions. This requires coordinating what information each query provides, what operations the tools expose, and what information and state transitions the simulator makes available at each step. Missing information or operations can make a task unsolvable. Conversely, revealing intermediate information prematurely—either in the query or through simulator responses before the required steps are completed—can allow the agent to bypass the intended dependencies.

As shown in Figure 1, consider a simple task that asks an agent to cancel a user's most recent unshipped order. The query provides the user's email and the order-selection criteria, but not the target `order_id`. The toolset must support identifying the user, retrieving their orders, and canceling the

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

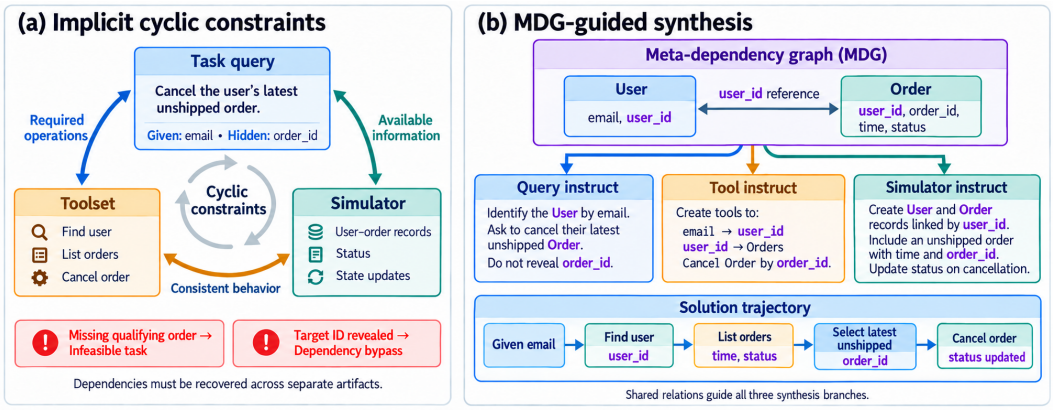

Figure 1: Overview of cyclic constraints in agentic environment synthesis and our MDG-guided solution. **Left:** synthesizing the query, toolset, and simulator as separate artifacts leaves their dependencies implicit, which can produce infeasible tasks or allow agents to bypass intended dependencies. **Right:** EnvFoundry exposes shared entity relations through an MDG and derives coordinated instructions for all three components.

selected order. The simulator must contain consistent user–order relations and a qualifying order, return the timestamps and shipping statuses needed for selection, and implement the cancellation operation. If no qualifying order exists, the task is infeasible. If the query directly supplies the target order identifier, the intended retrieval and selection steps may be bypassed. Thus, both **task feasibility** and **dependency preservation** depend on how all three components are constructed.

Existing methods commonly adopt sequential synthesis pipelines (Qin et al., 2023; Guo et al., 2024; Wang et al., 2026; Song et al., 2026; Dong et al., 2026), where later components must accommodate choices already fixed by earlier stages. When dependencies remain implicit, this requires recovering and reconciling requirements scattered across generated artifacts. Simplifying dependencies reduces this burden but limits task complexity (Castellani et al., 2025; Song et al., 2026). Post hoc execution-based filtering can reject invalid instances (Wang et al., 2026; Fang et al., 2025), but successful execution alone does not establish that the intended dependencies were preserved. What is missing is an explicit shared structure that coordinates these requirements during synthesis.

Our key insight is that these cyclic constraints arise in part from a shared structure of **entities and attribute relations**. Although queries, tools, and simulators serve different roles, they must describe compatible ways of accessing and manipulating the same underlying entities. In the example above, the user–order relation determines which orders the query refers to, how tools retrieve those orders, and how the simulator associates their records with a user. Preserving the intended dependencies further requires coordinating information availability: the query must leave the target order identifier to be discovered, the tools must provide a retrieval path, and the simulator must expose the required attributes through the appropriate interactions. Rather than reconstructing these requirements separately for each component, we make their shared structural basis explicit before synthesis.

Motivated by this insight, we propose EnvFoundry, a scalable coarse-to-fine framework for agentic environment synthesis (Figure 2). For each task topic, we first generate a *meta-dependency graph* (MDG), whose nodes represent entity types and whose edges encode dependencies between entity attributes through identifier references or shared attributes. The MDG captures reusable relational structure, which is then specialized into concrete task requirements, tool behaviors, and simulator states. Specifically, the MDG guides three complementary sets of synthesis instructions. **Query instructions** translate entity relations into cross-entity task requirements, specifying the selection criteria and information that the agent must discover. **Tool instructions** define how tools access related entities, expose attributes needed by subsequent actions, and perform state updates. **Simulator instructions** define the corresponding state schema and relational constraints, then specify task-specific records and response behavior that support the required interactions.

By grounding all three components in the same MDG, EnvFoundry converts implicit cross-component requirements into explicit, coordinated synthesis instructions. This design supports **con-**

sistency through shared structural constraints and execution-based validation, and **controllability** through explicit dependency requirements and their task-specific instantiation.

Intuitively, this process resembles solving a system of coupled equations: assigning one variable constrains the admissible values of the others, and fixing variables sequentially without accounting for their shared constraints can leave later equations unsatisfied. EnvFoundry instead reparameterizes the synthesis problem around a shared latent structure—the MDG. Task queries, toolsets, and simulators are constructed as complementary realizations of this structure, allowing their synthesis to proceed under a common set of relational constraints. This shifts the challenge from reconciling separately generated components to deriving structurally aligned components from a shared specification.

In summary, this paper makes the following contributions.

- We identify the cyclic constraints among queries, toolsets, and simulators that jointly govern task solvability and dependency preservation.
- We propose EnvFoundry, a from-scratch synthesis framework that takes only a domain name as input and uses an MDG to coordinate the synthesis of task queries, toolsets, and simulators.
- We demonstrate that EnvFoundry enables effective, low-cost environment synthesis, consistently improving performance across τ-bench, τ^2-Bench, and VitaBench, with gains of up to 21.14 percentage points.

2 Related Work

2.1 Tool Learning

Tool learning equips large language models (LLMs) with the ability to select appropriate tools, construct valid arguments, and incorporate execution results into subsequent decisions. Existing work spans supervised learning from tool-use demonstrations (Qin et al., 2023; Guo et al., 2025; Zhang et al., 2025; 2024), specialized representations and inference-time planning (Hao et al., 2023; Chen et al., 2024), and reinforcement learning for multi-step interaction (Shi et al., 2024; Lin et al., 2025; Feng et al., 2025; Li et al., 2025b; Dong et al., 2025b; Mai et al., 2025; Dong et al., 2025a). Collectively, these methods have advanced tool use from isolated API-call prediction toward sequential decision making over interactive trajectories.

Most tool-learning methods, however, treat the interaction environment as given, assuming that task queries, tool interfaces, backend states, and execution dynamics have already been constructed. Yet the quality and diversity of these environments directly determine the interactions and feedback available for training. Our work is complementary to this line of research: rather than improving the tool-use policy itself, we study how to construct coherent and executable environments that support scalable agent training and evaluation.

2.2 Tool-Use Data and Environments

The scarcity of real-world agentic data has made synthetic tool-use data and environments important resources for training agents (Qin et al., 2023; Qu et al., 2024). Early approaches often build on task queries or tool documentation collected from real-world sources and synthesize the remaining components (Guo et al., 2024; Ye et al., 2026; Wang et al., 2024; Zeng et al., 2025; Liu et al., 2024a).

More recent work constructs executable environments that support agent exploration, state transitions, and grounded feedback, improving their quality through fine-grained verification, real-world grounding, and environment-aware training (Castellani et al., 2025; Fang et al., 2025; Song et al., 2026; Cai et al., 2025; Tu et al., 2026; Liu et al., 2026; Huang et al., 2026). For example, AWM introduces a code-augmented LLM-as-a-Judge to assess synthesized environments beyond programmatic state checks (Wang et al., 2026), whereas Agent-World and EnvFactory use real-world Internet resources as seeds for synthesis (Dong et al., 2026; Xu et al., 2026).

Despite these advances, component generation generally follows a sequential pipeline, with task queries, toolsets, and simulators synthesized in an ordered sequence conditioned on previously gen-

erated artifacts. This formulation does not explicitly resolve the cyclic constraints among the three components that jointly determine task solvability and difficulty. Our work addresses this structural limitation by deriving coordinated synthesis instructions from a shared meta dependency graph, making these constraints explicit before component generation. This focus on the underlying dependencies is orthogonal to improvements in , verification, realistic grounding and training method, and can be combined with them.

3 METHODOLOGY

EnvFoundry synthesizes an agentic environment entirely from scratch. Its only required input is a domain name d, such as *retail* or *airline*; it assumes no pre-existing task queries, tool definitions, database schemas, or simulator states. Starting from this domain-level seed, the framework constructs the entity structure, tool interfaces and implementations, executable simulator, and task instances.

We view a tool-use environment as a triple $(q, \mathcal{T}, \mathcal{B})$, where q is a natural-language query, $\mathcal{T}$ is a toolset, and $\mathcal{B}$ is a structured simulator state. The toolset and simulator are defined as

$$\mathcal{T} = \{f_i\}_{i=1}^{N}, \qquad f_i = (\sigma_i, F_i), \qquad \mathcal{B} = (\Sigma, D_0). \tag{1}$$

Here, σ_i specifies a tool's name, description, argument schema, and return fields, while F_i is its executable implementation. The schema Σ defines entity types, typed attributes, and relational constraints, and D_0 is an initial state conforming to Σ. A tool call $a_t = (i_t, \mathbf{u}_t)$ selects f_{i_t} with arguments $\mathbf{u}_t$ and executes

$$F_{i_t}(D_t, \mathbf{u}_t) = (D_{t+1}, o_{t+1}), \tag{2}$$

where D_t is the database state at step t and o_{t+1} is the returned observation. Together, $\mathcal{T}$ and $\mathcal{B}$ form an executable environment in which the query is solved.

Figure 2 exhibits the full EnvFoundry pipeline. *Scaffold synthesis* infers coherent entity topics and constructs an MDG for each topic. *Instruction synthesis* specializes MDG relations into aligned query, tool, and simulator specifications. *Environment synthesis* instantiates, executes, and validates the three components under these shared specifications. A single MDG and toolset can support multiple task queries and initial states, enabling scalable generation without discarding coordinated control over individual task dependencies. The subsequent interaction-design stage determines how the verified environment is presented to the agent. Appendix A provides detailed statistics and analysis of the synthesized environments.

3.1 SCAFFOLD SYNTHESIS

Domain Topic Inference. Given a domain d, we first infer an entity pool $\mathcal{X}_d$ and group it into coherent topics $\{X_1, \ldots, X_M\}$, where $X_m \subseteq \mathcal{X}_d$. Entities may participate in multiple topics. Each topic defines the semantic scope of an environment independently of any particular tool interface or task query.

MDG Construction. For a topic X, we set $V = X$ and annotate each entity $x \in V$ with its attributes A_x. An edge $e \in E$ connects two attributes and specifies their relation:

$$e = (x.a,\, y.b,\, \rho), \qquad a \in A_x, \quad b \in A_y, \quad \rho \in \{\mathrm{ref}, \mathrm{shared}\}. \tag{3}$$

Here, ref denotes that one attribute stores an identifier of the other entity, while shared denotes semantically aligned values that support matching or comparison. The MDG is intentionally defined before concrete task and tool artifacts. Unlike a tool dependency graph, it therefore represents a space of possible information-access structures rather than dependencies among predefined actions.

3.2 INSTRUCTION SYNTHESIS

Given an MDG, we select connected subgraphs that can support meaningful state changes and synthesize a set $\mathcal{Z}$ of *dependency instructions* (DIs). A DI $z \in \mathcal{Z}$ contains an objective g_z, a decision rule r_z, and an evidence flow p_z. The objective specifies the desired state change. The rule identifies the target record or action arguments by filtering, comparing, or matching attributes across entities.

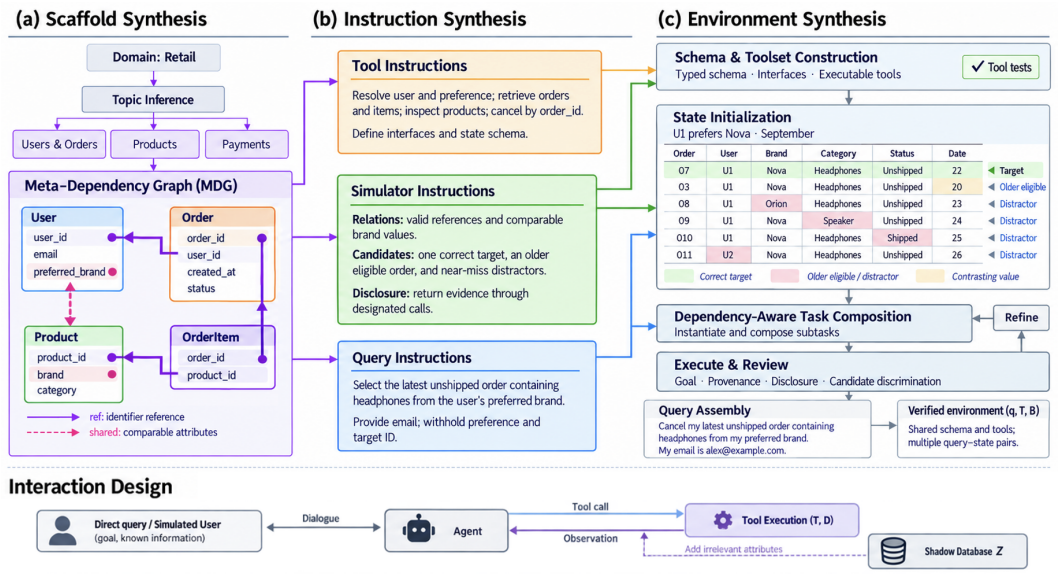

Figure 2: The EnvFoundry pipeline. Scaffold synthesis makes shared entity structure explicit through an MDG. Instruction synthesis derives coordinated query, tool, and simulator instructions. Environment synthesis instantiates, executes, reviews, and assembles the resulting environments and tasks.

The evidence flow specifies which relation supplies each required value and in what order that value becomes accessible. We then derive a coordinated instruction triple

$$\Gamma_z = \left(\gamma_z^{\mathcal{T}}, \gamma_z^{\mathcal{B}}, \gamma_z^{q}\right), \tag{4}$$

whose three projections constrain the corresponding environment components.

Tool Instructions. The tool instruction $\gamma_z^{\mathcal{T}}$ specifies the capabilities needed to realize p_z through the simplest tool interfaces that can complete the DI. Each tool performs a minimal operation, such as locating an initial entity, retrieving directly related records, or applying the final state change, rather than collapsing multiple dependency steps into one call. The instruction also specifies which attributes each response exposes and constrains the executable behavior accordingly. Every latent value required by a downstream call must be produced by an earlier permitted call, while no tool should accept or reveal information that bypasses the intended access path. The resulting toolset therefore makes the agent follow the DI through multiple rounds of tool use. Across DIs, these instructions induce read tools for identification and retrieval and write tools for creation, deletion, link modification, and domain-specific business actions.

Simulator Instructions. The simulator instruction $\gamma_z^{\mathcal{B}}$ has structural and instance-level parts. The structural part specifies entity attributes, the MDG relations they must preserve, and the state transitions associated with the instructed tools. The instance-level part yields database constraints $\mathcal{C}_z$: required records and links must exist, action preconditions must hold, and the database must contain sufficient distractor records to make the selection rule necessary. These distractors remain plausible candidates but differ on decision-relevant attributes, so that r_z resolves to one unambiguous target only after the required evidence has been collected. The instruction also specifies which fields are returned by each interaction, aligning simulator observations with the information schedule in p_z.

Query Instructions. The query instruction γ_z^{q} specifies the user-facing objective, selection criteria, and the minimal requester-supplied information $\mathcal{I}_0$ needed to begin the task. It requires the synthesized query to be as concise as possible: every stated detail must be necessary to determine the intended objective or initiate the evidence flow. All other information, including intermediate attributes and target identifiers determined by r_z, remains latent and must be obtained through tool interaction. The query therefore neither supplies redundant hints nor describes a procedure for solv-

ing the task. This projection turns an MDG relation into a cross-entity reasoning requirement rather than a procedural list of tool calls.

Together, these instructions provide fine-grained control over what the query withholds, what the tools reveal, and what the simulator instantiates. In our synthesis configuration, DIs are typically designed to induce multi-step interaction, with target trajectories often requiring 5–10 tool calls. We then compose multiple DIs into longer and more diverse dependency chains, enabling the scalable generation of large numbers of complex tasks.

3.3 Environment Synthesis

In the final synthesis stage, we compile the coordinated instructions into a reusable executable environment and task-specific query–state pairs.

Schema and Toolset. We construct Σ by assigning types to the attributes A_x of each entity $x \in V$ and translating MDG edges into reference and shared-attribute constraints. A database state has the form $D_t = \{D_t^x\}_{x \in V}$, where D_t^x contains records of type x. We then consolidate the tool instructions across DIs into $\mathcal{T} = \mathcal{T}_{\text{read}} \cup \mathcal{T}_{\text{write}}$. For each instructed operation, we generate an interface σ_i and an implementation F_i over Σ. Additional basic retrieval and update operations induced by MDG nodes and edges may be included to make the environment reusable across multiple tasks, but the fields returned by each tool remain explicit in its interface and implementation.

We test generated tools on states conforming to Σ, checking argument validation, returned observations, state updates, and relational integrity. Only schema–toolset pairs that pass these checks are retained.

Scenario State Initialization. For a selected set of DIs, the schema constraints $\mathcal{C}_\Sigma$ and task-specific simulator constraints determine the initial state:

$$\mathcal{C} = \mathcal{C}_\Sigma \cup \bigcup_{z \in \mathcal{Z}} \mathcal{C}_z, \qquad D_0 \models \mathcal{C}. \tag{5}$$

We generate D_0 conditioned on $\mathcal{C}$ and verify that it contains both the evidence required by each decision rule and sufficient variation to make that evidence necessary. This produces the task-specific simulator $\mathcal{B} = (\Sigma, D_0)$ while preserving a schema shared across many instances.

Dependency-Aware Task Composition. We instantiate selected DIs as subtasks in D_0, preserving their objectives, decision rules, evidence flows, and candidate scopes. Multiple subtasks may be composed as $S = (s_1, \ldots, s_K)$ under one coherent user goal. Let $\mathcal{I}_k$ be the information required by s_k, $\mathcal{I}_0$ the requester-supplied information, $\mathcal{F}_j$ the facts established by s_j, and $\mathcal{R}_k$ the information retrievable through the intended tool path in the current state. We require

$$\mathcal{I}_k \subseteq \mathcal{I}_0 \cup \bigcup_{j<k} \mathcal{F}_j \cup \mathcal{R}_k. \tag{6}$$

For each intended dependency, at least one required value is excluded from $\mathcal{I}_0$ and assigned by the evidence flow to a preceding observation or state change. Equation 6 ensures feasibility, while this non-disclosure condition prevents the query from collapsing the intended multi-step structure. Composition can also make a later DI depend on facts or state changes produced by an earlier DI.

Execution, Review, and Query Assembly. Starting from D_0, a solver executes S using $\mathcal{T}$, and a reviewer checks goal completion, the provenance of each action argument, consistency between observations and state, and unnecessary disclosure of retrievable information. Ambiguous, infeasible, or insufficiently grounded subtasks are refined and re-executed. Accepted tasks provide a reference trajectory $\xi^* = ((a_t, o_{t+1}))_{t=0}^{T-1}$ and its terminal state D_T^*. We then realize the query instructions as q, expressing the verified goals and user-known conditions while removing the internal plan and latent values. The final output $(q, \mathcal{T}, \mathcal{B})$ therefore satisfies *query–environment complementarity*: the query states what the user wants and what the user knows, whereas the environment supplies the intermediate evidence and state transitions needed to accomplish it.

3.4 INTERACTION DESIGN

With $(q, \mathcal{T}, \mathcal{B})$ fixed, interaction design determines how task-relevant information reaches the agent through tools and user dialogue. We model interaction as a partially observable decision process: the agent receives tool interfaces $\{\sigma_i\}_{i=1}^{N}$ and observes tool responses and dialogue, while D_t remains internal to the environment. Given the interaction history H_t, the agent policy π selects $a_t \sim \pi(\cdot \mid H_t)$. The selected tool implementation F_{i_t} produces the next database state and observation according to Equation 2.

Shadow Database. Tool use involves two complementary challenges: executing state-changing actions with correctly composed arguments and extracting relevant evidence from retrieval observations. Dependency-aware synthesis captures the former through cross-entity dependencies and state transitions, but task-oriented databases may underrepresent the latter by exposing predominantly relevant information. Adding irrelevant attributes to the state schema Σ would increase synthesis and validation costs, coupling observation complexity with the complexity of the underlying environment. We therefore introduce a *shadow database*, a general mechanism for scaling observation complexity independently of state dynamics.

After environment synthesis, we generate semantically plausible but task-irrelevant attributes for each searchable record and store them separately in Z. During interaction, the simulator augments retrieval observations as

$$\widetilde{o}_{t+1} = \mathrm{Augment}(o_{t+1}; Z). \tag{7}$$

This augmentation preserves all original fields and values, while tool execution continues to operate on the true state D_t according to Equation 2. The true database thus remains the authoritative source for task-relevant information and state transitions, while the shadow database enriches the observation layer. Varying the amount and diversity of shadow attributes controls the information extraction burden without expanding the core schema or altering task semantics. This separation provides a reusable way to increase retrieval difficulty in synthesized environments while preserving their verified execution behavior.

Simulated User. When generating conversational trajectories for SFT, we simulate user interaction so that goals and conditions can be revealed over multiple turns. We express the verified task as a hidden goal g and user-known information $\mathcal{I}_0$. An LLM-based user simulator generates a message

$$m_{t+1} \sim p_{\mathrm{user}}(\cdot \mid g, \mathcal{I}_0, H_t), \tag{8}$$

progressively disclosing information as the conversation requires. For RL training, in contrast, we provide the assembled query q at initialization and let the agent complete the task directly, without invoking the user simulator. Both training formats retain the verified task's intended outcome and use the same $\mathcal{T}$ and $\mathcal{B}$.

4 EXPERIMENTS

Experiments on τ-bench, τ^2-Bench, and VitaBench demonstrate that EnvFoundry substantially improves tool-use performance, with gains of up to 21.14 percentage points across these benchmarks. Our trained 4B model outperforms all evaluated baselines trained with other environment synthesis frameworks, including a 14B model, across all three benchmarks. Ablation studies highlight the importance of MDG-derived instructions for coordinating the synthesis of queries, toolsets, and simulators, along with the benefits of the shadow database for improving environment quality. MDG-guided synthesis also keeps construction inexpensive by focusing generation on the data and dependencies needed to ensure task solvability and the intended difficulty, at an estimated API cost of only \$0.61 per environment.

Benchmarks. We evaluate on τ-bench (Yao et al., 2024), τ^2-Bench (Barres et al., 2025), and VitaBench (He et al., 2025) (abbreviated as Vita in the tables). We report task success rates across the following domains: Retail and Airline for τ-bench; Retail, Airline, and Telecom for τ^2-Bench; and Delivery, Online Travel Agency (OTA), and In-store Consumption for VitaBench. For each benchmark, Overall denotes the unweighted mean of the domain-level success rates. Further benchmark details are provided in Appendix C.1.

Table 1: Main results on τ-bench, τ^2-Bench, and VitaBench. Task success rates are reported as percentages. Overall is the unweighted mean across the domains of each benchmark. Subscripts beside the Overall scores of EnvFoundry report absolute gains over the corresponding Qwen3 checkpoint.

Model	τ-bench			τ^2-Bench				Vita			
	Retail	Airline	Overall	Retail	Airline	Telecom	Overall	Delivery	OTA	Instore	Overall
AWM-4B	33.04	22.00	27.52	28.07	24.00	21.10	24.39	16.66	6.33	19.33	14.11
AWM-8B	44.35	28.00	36.18	43.86	32.67	19.01	31.85	18.67	13.00	27.67	19.78
AWM-14B	49.28	30.00	39.64	50.59	31.33	22.52	34.81	29.67	17.33	39.48	28.83
EnvScaler-4B	48.99	30.67	39.83	45.61	27.33	21.93	31.62	28.44	4.00	24.33	18.92
EnvScaler-8B	54.20	30.67	42.44	54.39	29.33	20.18	34.63	27.00	11.33	38.00	25.44
Qwen3-4B (Thinking)	35.94	26.00	**30.97**	31.87	23.33	17.40	**24.20**	15.33	6.33	27.33	**16.33**
+ EnvFoundry (w/ SFT)	58.55	30.00	**44.28**$_{+13.31}$	59.06	28.67	34.21	**40.65**$_{+16.45}$	28.67	10.00	33.33	**24.00**$_{+7.67}$
+ EnvFoundry (w/ SFT&RL)	61.16	32.67	**46.92**$_{+15.95}$	62.87	37.33	34.21	**44.80**$_{+20.60}$	31.33	15.67	44.67	**30.56**$_{+14.23}$
Qwen3-30B-A3B (Thinking)	37.97	29.33	**33.65**	45.61	35.33	17.25	**32.73**	25.67	13.67	27.67	**22.34**
+ EnvFoundry (w/ SFT)	64.64	39.33	**51.99**$_{+18.34}$	65.79	44.67	32.02	**47.49**$_{+14.76}$	33.00	19.33	50.67	**34.33**$_{+11.99}$
+ EnvFoundry (w/ SFT&RL)	69.57	40.00	**54.79**$_{+21.14}$	69.59	49.33	42.52	**53.81**$_{+21.08}$	35.00	19.33	52.00	**35.44**$_{+13.10}$

Table 2: Ablations of dependency-modeling components on Qwen3-4B. Each variant is evaluated after SFT on 4,000 randomly selected training samples.

Variant	τ-bench		τ^2-Bench			Avg.
	Retail	Airline	Retail	Airline	Telecom	
w/o MDG	39.42	14.00	46.78	16.67	12.85	25.94
w/o dependency instructions	41.74	17.33	42.98	18.33	13.60	26.80
w/o Simulator instructions	41.45	12.00	44.44	20.67	17.31	27.17
Full pipeline	**58.26**	**30.00**	**60.23**	**26.67**	**19.63**	**38.96**

Baselines. We use Qwen3-4B (Thinking) and Qwen3-30B-A3B (Thinking) as backbone models and include their initial checkpoints as baselines. We also compare with agents trained in synthetic environments using AWM (Wang et al., 2026) with Qwen3 backbones of 4B, 8B, and 14B parameters, and EnvScaler (Song et al., 2026) with Qwen3 backbones of 4B and 8B parameters.

Main Results. Training with EnvFoundry consistently improves performance across all three benchmarks and both model scales, with RL providing further gains beyond SFT (Table 1). For Qwen3-4B, SFT increases the Overall scores by 13.31, 16.45, and 7.67 percentage points on τ-bench, τ^2-Bench, and Vita, respectively, surpassing the size-matched AWM-4B and EnvScaler-4B baselines on every benchmark. Further RL training extends these gains to 15.95, 20.60, and 14.23 points. The resulting 4B SFT&RL model outperforms all reported AWM and EnvScaler variants in Overall score across the three benchmarks, including those with larger parameter counts. On τ^2-Bench, it exceeds the strongest of these baselines by 9.99 points. Qwen3-30B-A3B also benefits substantially from EnvFoundry. SFT improves the Overall scores by 18.34, 14.76, and 11.99 points on the three benchmarks, respectively, while SFT&RL extends these gains to 21.14, 21.08, and 13.10 points. The final model achieves Overall scores of 54.79 on τ-bench, 53.81 on τ^2-Bench, and 35.44 on Vita. The consistent gains from RL at both model scales demonstrate that the synthesized environments support both trajectory-based supervision and continued learning through interaction.

Dependency Modeling Ablations. Effective dependency modeling requires coordinated MDG-derived guidance across the query, toolset, and simulator; omitting even one of the guidances substantially weakens the full pipeline. We evaluate this coordination through ablations on Qwen3-4B using τ-bench and τ^2-Bench, reporting the unweighted mean success rate across their five domains. As shown in Table 2, the full pipeline achieves 38.96, whereas removing the MDG lowers performance to 25.94 (-13.02 points), highlighting the importance of a shared dependency scaffold. Retaining the MDG but removing dependency instructions reduces performance to 26.80 (-12.16 points), showing that the scaffold must be translated into task-level constraints to guide component synthesis. Even when the MDG and dependency instructions are retained, removing simulator instructions lowers performance to 27.17 (-11.79 points). This result highlights the need for

Table 3: Effect of shadow observation augmentation on Qwen3-4B. Each variant is evaluated after SFT on 4,000 randomly selected training samples.

Variant	τ-bench		τ^2-Bench			Avg.
	Retail	Airline	Retail	Airline	Telecom	
Shadow disabled	54.20	16.67	51.75	23.33	**23.89**	33.98
Shadow enabled	**58.26**	**30.00**	**60.23**	**26.67**	19.63	**38.96**

Table 4: Zero-RL results on Qwen3-4B and Qwen3-30B-A3B.

Model	τ-bench		τ^2-Bench			Avg.
	Retail	Airline	Retail	Airline	Telecom	
Qwen3-4B (Thinking)	**35.94**	26.00	31.87	23.33	**17.40**	26.91
+ EnvFoundry (Zero-RL)	35.36	**29.33**	**43.28**	**24.67**	11.84	**28.90**
Qwen3-30B-A3B (Thinking)	37.97	29.33	45.61	35.33	17.25	33.10
+ EnvFoundry (Zero-RL)	**47.83**	**30.00**	**59.65**	**40.00**	**20.18**	**39.53**

MDG-derived guidance to cover the synthesis of all three components: leaving even one component unguided can compromise environment quality and substantially reduce downstream performance.

Shadow Database. The shadow database improves overall downstream performance by enriching retrieval observations without changing the underlying tasks or tool schema. As shown in Table 3, enabling it increases Avg. from 33.98 to 38.96, a gain of 4.98 percentage points. This result suggests that training agents to extract relevant evidence from richer observations improves their tool-use capabilities. The benefits vary across domains, however, with Telecom showing a decrease in this setting.

Zero-RL. Direct RL on the synthesized environments improves overall performance for both model scales, but is less effective than RL preceded by SFT. We refer to training directly from the initial checkpoints without an SFT stage as *Zero-RL*. As shown in Table 4, Zero-RL increases the five-domain Avg. across the two τ benchmarks by 1.99 points for Qwen3-4B and 6.43 points for Qwen3-30B-A3B. The larger model improves in every domain, whereas the smaller model exhibits mixed domain-level changes. This gap suggests that the benefits of direct RL depend on the model's initial capabilities: weaker models may struggle to generate successful trajectories that provide informative learning signals. SFT helps establish the tool-use capabilities needed for subsequent exploration, enabling more effective RL.

5 Conclusion and Discussion

In this work, We present EnvFoundry, a scalable framework for agentic environment synthesis that addresses the cyclic constraints among task queries, toolsets, and simulators. By deriving coordinated synthesis instructions from a shared meta-dependency graph, EnvFoundry aligns these components while preserving the dependencies required for multi-step interaction. Experiments on τ-bench, τ^2-Bench, and VitaBench demonstrate its effectiveness for both supervised fine-tuning and reinforcement learning.

However, Agentic environment synthesis is a systems engineering challenge with many opportunities for optimization. Our contribution centers on explicit dependency modeling to coordinate the synthesis of queries, toolsets, and simulators, rather than an exhaustive optimization of the entire pipeline. Complementary advances in verification (Wang et al., 2026), reinforcement learning with simulated users (Cai et al., 2025), and realistic grounding (Dong et al., 2026; Song et al., 2026) have demonstrated substantial benefits for agent training. These advances can be combined with EnvFoundry, offering further opportunities to improve environment quality and downstream performance.

AI USE STATEMENT

The research ideas, problem formulation, methodology, experimental design and execution, data analysis, interpretation of results, and manuscript preparation were primarily carried out by the authors. Generative AI tools were used solely in an assistive capacity. All AI-generated or AI-assisted content was independently reviewed, verified, and revised by the authors, who take full responsibility for the accuracy, originality, and integrity of the final work.

REFERENCES

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. τ^2-bench: Evaluating conversational agents in a dual-control environment. *arXiv preprint arXiv:2506.07982*, 2025.

Shihao Cai, Runnan Fang, Jialong Wu, Baixuan Li, Xinyu Wang, Yong Jiang, Liangcai Su, Liwen Zhang, Wenbiao Yin, Zhen Zhang, Fuli Feng, Pengjun Xie, and Xiaobin Wang. AutoForge: Automated environment synthesis for agentic reinforcement learning. *arXiv preprint arXiv:2512.22857*, 2025.

Tommaso Castellani, Naimeng Ye, Daksh Mittal, Thomson Yen, Emmanouil Koukoumidis, William Zeng, and Hongseok Namkoong. SynthTools: A framework for scaling synthetic tools for agent development. *arXiv preprint arXiv:2511.09572*, 2025.

Sijia Chen, Yibo Wang, Yi-Feng Wu, Qing-Guo Chen, Zhao Xu, Weihua Luo, Kaifu Zhang, and Lijun Zhang. Advancing tool-augmented large language models: Integrating insights from errors in inference trees. *arXiv preprint arXiv:2406.07115*, 2024.

DeepSeek-AI, Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, et al. DeepSeek-V3.2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*, 2025.

Guanting Dong, Licheng Bao, Zhongyuan Wang, Kangzhi Zhao, Xiaoxi Li, Jiajie Jin, Jinghan Yang, Hangyu Mao, Fuzheng Zhang, Kun Gai, Guorui Zhou, Yutao Zhu, Ji-Rong Wen, and Zhicheng Dou. Agentic entropy-balanced policy optimization. *arXiv preprint arXiv:2510.14545*, 2025a.

Guanting Dong, Yifei Chen, Xiaoxi Li, Jiajie Jin, Hongjin Qian, Yutao Zhu, Hangyu Mao, Guorui Zhou, Zhicheng Dou, and Ji-Rong Wen. Tool-Star: Empowering LLM-brained multi-tool reasoner via reinforcement learning. *arXiv preprint arXiv:2505.16410*, 2025b.

Guanting Dong, Junting Lu, Junjie Huang, Wanjun Zhong, Longxiang Liu, Shijue Huang, Zhenyu Li, Yang Zhao, Xiaoshuai Song, Xiaoxi Li, Jiajie Jin, Yutao Zhu, Hanbin Wang, Fangyu Lei, Qinyu Luo, Mingyang Chen, Zehui Chen, Jiazhan Feng, Ji-Rong Wen, and Zhicheng Dou. AgentWorld: Scaling real-world environment synthesis for evolving general agent intelligence. *arXiv preprint arXiv:2604.18292*, 2026.

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

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

Zhicheng Guo, Sijie Cheng, Hao Wang, Shihao Liang, Yujia Qin, Peng Li, Zhiyuan Liu, Maosong Sun, and Yang Liu. StableToolBench: Towards stable large-scale benchmarking on tool learning of large language models. *arXiv preprint arXiv:2403.07714*, 2024.

Zhicheng Guo, Sijie Cheng, Yuchen Niu, Hao Wang, Sicheng Zhou, Wenbing Huang, and Yang Liu. StableToolBench-MirrorAPI: Modeling tool environments as mirrors of 7,000+ real-world APIs. *arXiv preprint arXiv:2503.20527*, 2025.

Shibo Hao, Tianyang Liu, Zhen Wang, and Zhiting Hu. ToolkenGPT: Augmenting frozen language models with massive tools via tool embeddings. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

Wei He, Yueqing Sun, Hongyan Hao, Xueyuan Hao, Zhikang Xia, Qi Gu, Chengcheng Han, Dengchang Zhao, Hui Su, Kefeng Zhang, Man Gao, Xi Su, Xiaodong Cai, Xunliang Cai, Yu Yang, and Yunke Zhao. VitaBench: Benchmarking LLM agents with versatile interactive tasks in real-world applications. *arXiv preprint arXiv:2509.26490*, 2025.

Chengsong Huang, Zifeng Wang, Rujun Han, Jun Yan, Yanfei Chen, Zoey CuiZhu, Ke Jiang, Peng Xia, Han Yu, Yufan Zhuang, Yifei Ming, Jiaqi Pan, Bhavana Dalvi Mishra, Jiaxin Huang, Burak Gokturk, Tomas Pfister, and Chen-Yu Lee. EnvHarness: Awakening static worlds for agent learning. *arXiv preprint arXiv:2608.19880*, 2026.

Kimi Team, Yifan Bai, Yiping Bao, Y. Charles, Cheng Chen, Guanduo Chen, Haiting Chen, Huarong Chen, Jiahao Chen, Ningxin Chen, et al. Kimi K2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*, 2025.

Kuan Li, Zhongwang Zhang, Huifeng Yin, Liwen Zhang, Litu Ou, Jialong Wu, Wenbiao Yin, Baixuan Li, Zhengwei Tao, Xinyu Wang, Weizhou Shen, Junkai Zhang, Dingchu Zhang, Xixi Wu, Yong Jiang, Ming Yan, Pengjun Xie, Fei Huang, and Jingren Zhou. WebSailor: Navigating super-human reasoning for web agent. *arXiv preprint arXiv:2507.02592*, 2025a.

Xuefeng Li, Haoyang Zou, and Pengfei Liu. ToRL: Scaling tool-integrated RL. *arXiv preprint arXiv:2503.23383*, 2025b.

Zihan Lin, Xiaohan Wang, Jie Cao, Jiajun Chai, Guojun Yin, Wei Lin, and Ran He. ResT: Reshaping token-level policy gradients for tool-use large language models. *arXiv preprint arXiv:2509.21826*, 2025.

Bo Liu, Simon Yu, Yiding Jiang, Ao Qu, Andrew Zhao, Zichen Liu, Junsu Kim, Zijian Zhou, Seungone Kim, Tongzheng Ren, Mickel Liu, Hanfei Yu, Zhaorun Chen, Weiyan Shi, Paul Pu Liang, Luke Zettlemoyer, Yejin Choi, and Natasha Jaques. SPADE: Self-play in adaptive synthetic executable environments. *arXiv preprint arXiv:2608.19197*, 2026.

Weiwen Liu, Xu Huang, Xingshan Zeng, Xinlong Hao, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Zhengying Liu, Yuanqing Yu, Zezhong Wang, Yuxian Wang, Wu Ning, Yutai Hou, Bin Wang, Chuhan Wu, Xinzhi Wang, Yong Liu, Yasheng Wang, Duyu Tang, Dandan Tu, Lifeng Shang, Xin Jiang, Ruiming Tang, Defu Lian, Qun Liu, and Enhong Chen. ToolACE: Winning the points of LLM function calling. *arXiv preprint arXiv:2409.00920*, 2024a.

Zuxin Liu, Thai Hoang, Jianguo Zhang, Ming Zhu, Tian Lan, Shirley Kokane, Juntao Tan, Weiran Yao, Zhiwei Liu, Yihao Feng, Rithesh Murthy, Liangwei Yang, Silvio Savarese, Juan Carlos Niebles, Huan Wang, Shelby Heinecke, and Caiming Xiong. APIGen: Automated pipeline for generating verifiable and diverse function-calling datasets. *arXiv preprint arXiv:2406.18518*, 2024b.

Xinji Mai, Haotian Xu, Zhong-Zhi Li, Xing W, Weinong Wang, Jian Hu, Yingying Zhang, and Wenqiang Zhang. Agent RL scaling law: Agent RL with spontaneous code execution for mathematical problem solving. *arXiv preprint arXiv:2505.07773*, 2025.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive APIs. *arXiv preprint arXiv:2305.15334*, 2023.

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

Cheng Qian, Zuxin Liu, Akshara Prabhakar, Zhiwei Liu, Jianguo Zhang, Haolin Chen, Heng Ji, Weiran Yao, Shelby Heinecke, Silvio Savarese, Caiming Xiong, and Huan Wang. UserBench: An interactive gym environment for user-centric agents. *arXiv preprint arXiv:2507.22034*, 2025.

Zile Qiao, Guoxin Chen, Xuanzhong Chen, Donglei Yu, Wenbiao Yin, Xinyu Wang, Zhen Zhang, Baixuan Li, Huifeng Yin, Kuan Li, Rui Min, Minpeng Liao, Yong Jiang, Pengjun Xie, Fei Huang, and Jingren Zhou. WebResearcher: Unleashing unbounded reasoning capability in long-horizon agents. *arXiv preprint arXiv:2509.13309*, 2025.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. *arXiv preprint arXiv:2307.16789*, 2023.

Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. Tool learning with large language models: A survey. *arXiv preprint arXiv:2405.17935*, 2024.

Wentao Shi, Mengqi Yuan, Junkang Wu, Qifan Wang, and Fuli Feng. Direct multi-turn preference optimization for language agents. *arXiv preprint arXiv:2406.14868*, 2024.

Xiaoshuai Song, Haofei Chang, Guanting Dong, Yutao Zhu, Ji-Rong Wen, and Zhicheng Dou. EnvScaler: Scaling tool-interactive environments for LLM agent via programmatic synthesis. *arXiv preprint arXiv:2601.05808*, 2026.

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

Hongru Wang, Rui Wang, Boyang Xue, Heming Xia, Jingtao Cao, Zeming Liu, Jeff Z. Pan, and Kam-Fai Wong. AppBench: Planning of multiple APIs from various APPs for complex user instruction. *arXiv preprint arXiv:2410.19743*, 2024.

Zhaoyang Wang, Canwen Xu, Boyi Liu, Yite Wang, Siwei Han, Zhewei Yao, Huaxiu Yao, and Yuxiong He. Agent world model: Infinity synthetic environments for agentic reinforcement learning. *arXiv preprint arXiv:2602.10090*, 2026.

Minrui Xu, Zilin Wang, Mengyi Deng, Zhiwei Li, Zhicheng Yang, Xiao Zhu, Yinhong Liu, Boyu Zhu, Baiyu Huang, Chao Chen, Heyuan Deng, Fei Mi, Lifeng Shang, Xingshan Zeng, and Zhijiang Guo. EnvFactory: Scaling tool-use agents via executable environments synthesis and robust RL. *arXiv preprint arXiv:2605.18703*, 2026.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*, 2024.

Junjie Ye, Changhao Jiang, Zhengyin Du, Yufei Xu, Xuesong Yao, Zhiheng Xi, Xiaoran Fan, Qi Zhang, Tao Gui, Xuanjing Huang, and Jiecao Chen. Feedback-driven tool-use improvements in large language models via automated build environments. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 2293–2323. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.findings-acl.109.

Xingshan Zeng, Weiwen Liu, Xu Huang, Zezhong Wang, Lingzhi Wang, Liangyou Li, Yasheng Wang, Lifeng Shang, Xin Jiang, Ruiming Tang, and Qun Liu. ToolACE-R: Model-aware iterative training and adaptive refinement for tool learning. *arXiv preprint arXiv:2504.01400*, 2025.

Jianguo Zhang, Tian Lan, Ming Zhu, Zuxin Liu, Thai Hoang, Shirley Kokane, Weiran Yao, Juntao Tan, Akshara Prabhakar, Haolin Chen, Zhiwei Liu, Yihao Feng, Tulika Awalgaonkar, Rithesh Murthy, Eric Hu, Zeyuan Chen, Ran Xu, Juan Carlos Niebles, Shelby Heinecke, Huan Wang, Silvio Savarese, and Caiming Xiong. xLAM: A family of large action models to empower AI agent systems. *arXiv preprint arXiv:2409.03215*, 2024.

Shaokun Zhang, Yi Dong, Jieyu Zhang, Jan Kautz, Bryan Catanzaro, Andrew Tao, Qingyun Wu, Zhiding Yu, and Guilin Liu. Nemotron-Research-Tool-N1: Tool-using language models with reinforced reasoning. *Preprints*, 2025. doi: 10.20944/preprints202504.2471.v1.

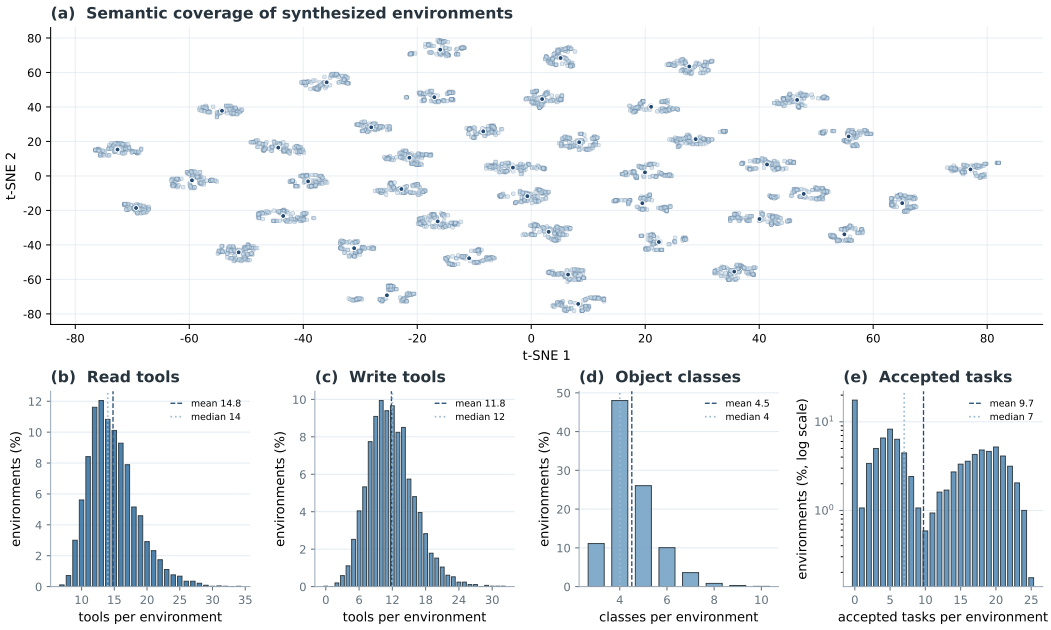

Figure 3: Environment diversity and task coverage. Panel (a) displays semantic coverage across the synthesized domains; dark markers denote domain centroids. Panels (b)–(d) show the distributions of read tools, write tools, and object classes per environment. Panel (e) compares candidate and accepted task counts per environment on a logarithmic vertical axis. The broad domain coverage, non-degenerate structural distributions, and long-tail task counts indicate that the corpus contains both reusable regularities and varied interaction complexity rather than repeated single-pattern instances.

A Environment Statistics and Analysis

The training corpus contains 2,276 SFT environments and 4,599 RL environments, with no overlap between the two pools. Both pools cover the same 42 canonical domains. Table 5 summarizes the environment, domain-label, and task counts. The input files contain 12,262 SFT tasks and 12,207 RL tasks; after complexity filtering, the RL loader retains 8,885 tasks.

Toolset and interface complexity. The environments provide substantial and varied tool sets. As shown in Figure 4(c), each environment contains 26.6 tools on average (median: 26), with a long tail extending beyond 50 tools. These comprise an average of 14.8 read tools and 11.8 write tools (Figure 3(b–c)), supporting both information retrieval and state modification. Agents therefore operate over sizable action spaces that combine state inspection with changes to the environment. Tool interfaces also vary in complexity: Figure 4(d) shows that, although tools with one or two parameters are the majority, the corpus also includes many tools with six to ten parameters and a small number with eleven. Together, these distributions highlight the demands on both tool selection and argument construction.

Long-horizon interaction. Figure 4(e) reports interaction steps and tool calls per accepted conversation. Both distributions peak at approximately 15–20, showing that typical trajectories already involve sustained multi-step interaction. Their extended tails reveal substantially longer cases: interaction lengths exceed 50 steps, while tool-call counts extend beyond 100 and, in an extreme case, exceed 200. These observed trajectories demonstrate that the corpus supports long-horizon tool use, in which agents must track intermediate results and evolving environment states across many actions. The distinction between interaction steps and tool calls also captures variation in tool-use intensity across conversations.

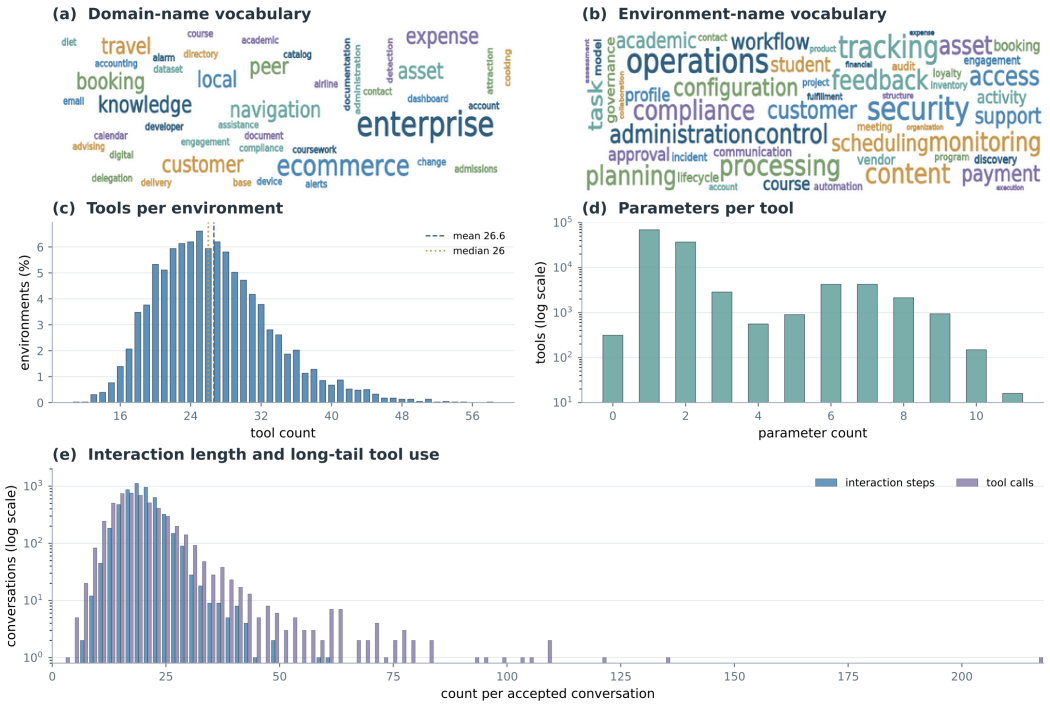

Figure 4: Tool-interface and accepted-trajectory statistics. Panels (a) and (b) show word clouds derived from domain and environment names, respectively. Panels (c) and (d) show tool counts per environment and parameter counts per tool; the latter uses a logarithmic vertical axis. Panel (e) compares interaction steps and tool calls over the accepted conversations in the measured corpus using the complete observed range and a logarithmic vertical axis to expose the long tail. The simultaneous concentration around typical toolset sizes and the extended tail of valid trajectories provide evidence that the environments support both routine workflows and multi-step cases.

Table 5: Training-corpus scope for SFT and RL. The environment pools are disjoint. Domain-label counts reflect the labels in the training records: SFT includes 15 historical labels beyond the 42 canonical domains.

Training pool	Environments	Domain labels	Tasks
SFT	2,276	57	12,262
RL	4,599	42	8,885

Validity Analysis We apply validation at environment, task, and trajectory level. A candidate must pass executable-tool tests, satisfy database constraints, produce a reachable reference path, and match the final-state checks. Rejection categories include implementation errors, schema or relation inconsistencies, unsatisfied constraints, unreachable target tools, invalid trajectories, and runtime exceptions.

Cost Analysis. MDG-guided synthesis keeps environment construction inexpensive by focusing generation on the data and dependencies needed to ensure task solvability and the intended difficulty. Rather than exhaustively populating a domain, the framework synthesizes records, attributes, and distractors according to explicit dependency constraints. As shown in Table 7, synthesis consumes approximately 6.27 million input tokens and 0.89 million output tokens per environment, with 3.75 million input tokens served from cache. The estimated API cost is only $0.61 per environment. These results illustrate how explicit structural control supports low-cost synthesis of task-relevant environments.

Table 6: Compact validity and rejection summary. The denominators and full failure taxonomy are maintained with the released analysis records.

Validation unit	Acceptance summary
SFT trajectory	State-validated acceptance: 30.97%
RL filtering instance	Kept when at least one of three rollouts is correct; observed keep rate: 80.8%

Table 7: Token consumption and estimated API cost per environment. Token counts are in thousands (k); cache hits are included in input tokens. Totals and costs are computed from the reported input/output counts before display rounding.

Stage	Input (k)	Cache hit (k)	Output (k)	Total (k)	Cost ($)
Scenario	2.20	0.00	8.00	10.20	0.0025
Task	364.60	256.00	261.40	626.00	0.0891
Database	151.40	78.60	100.20	251.60	0.0385
GT Annotation	5,593.80	3,400.40	432.60	6,026.40	0.4377
Toolset Schema	20.60	4.00	14.40	35.00	0.0064
Env Code	27.80	13.40	17.40	45.20	0.0069
Verification	108.60	2.20	59.44	168.04	0.0315
Total	6,269.00	3,754.60	893.44	7,162.44	0.6127

B Distinction between MDG and Tool Dependency Graphs

Many existing environment-synthesis methods organize task generation around a *tool dependency graph*, where dependencies are defined over a predefined set of tools (Dong et al., 2026; Fang et al., 2025). While convenient, this formulation fixes the action space before the query and simulator are constructed, thereby constraining the other components around an already specified toolset. Moreover, tool-level dependencies are inherently procedural: sampling a path over the graph tends to specify which tools should be used and in what order. This limits the diversity of interaction structures, such as repeatedly invoking the same tool, comparing multiple candidates, or requiring a complex sequence of operations that is only implicitly expressed by a high-level user request.

EnvFoundry instead constructs the MDG before concrete queries, tools, or simulator instances are synthesized. The MDG models dependencies among entities and attributes rather than dependencies among actions, and therefore specifies *what information must be obtained* without prescribing *how it must be obtained*. Concrete tools, simulator states, and task requirements are then jointly derived from this shared structure. As a result, EnvFoundry can generate goal-level instructions whose underlying execution may involve repeated calls, comparison, filtering, branching, and other multi-step interactions, without explicitly exposing the execution procedure to the agent.

C Experimental Details

C.1 Evaluation Benchmarks

We evaluate on τ-bench, τ^2-Bench, and VitaBench, covering eight benchmark subsets. Table **??** summarizes their task counts and interaction budgets. We follow the official evaluation metrics and procedures of each benchmark, assessing task success using the official environment checkers for τ-bench and τ^2-Bench and the official GPT-4.1-based trajectory evaluator for VitaBench. The benchmark environments are separate from the synthetic training environments, with no shared task instances or trajectories.

C.2 SFT and RL Configuration

Supervised fine-tuning. We perform full-parameter SFT using the conversation prompt format throughout training. Table 9 reports the configurations for Qwen3-4B and Qwen3-30B-A3B. Both

Table 8: Evaluation benchmarks, task counts, and interaction budgets. Each subset is evaluated in three independent runs over the complete task set, with one trial per task in each run. GPT-4.1 serves as the user simulator for all benchmarks.

Benchmark	Subset	Tasks	Max steps	Success assessment
τ-bench	Retail	115	30	Environment checker
τ-bench	Airline	50	30	Environment checker
τ^2-Bench	Retail	114	40	Environment checker
τ^2-Bench	Airline	50	40	Environment checker
τ^2-Bench	Telecom	114	200	Environment checker
VitaBench	Delivery	100	300	GPT-4.1 trajectory evaluator
VitaBench	OTA	100	300	GPT-4.1 trajectory evaluator
VitaBench	In-store	100	300	GPT-4.1 trajectory evaluator

models are trained for three epochs with a learning rate of 10^{-5}, a cosine learning-rate schedule, and a global batch size of 64. We use DeepSpeed ZeRO-3, bfloat16 precision, and FlashAttention, with a maximum sequence length of 32,000 tokens. The runs use two and eight GPUs, respectively, with gradient accumulation adjusted to maintain the same global batch size.

Table 9: Supervised fine-tuning configurations.

Setting	Qwen3-4B	Qwen3-30B-A3B
Parameter updates	Full-parameter	Full-parameter
Distributed training	DeepSpeed ZeRO-3	DeepSpeed ZeRO-3
Numerical precision	bfloat16	bfloat16
Attention implementation	FlashAttention	FlashAttention
Number of GPUs	2	8
Training epochs	3	3
Learning rate	10^{-5}	10^{-5}
Maximum sequence length	32,000	32,000
Batch size per device	1	1
Gradient accumulation steps	32	8
Global batch size	64	64
Learning-rate schedule	Cosine	Cosine
Validation split	None	None
Optimizer steps	576	576

Reinforcement learning. We train Qwen3-4B and Qwen3-30B-A3B using GSPO. Rollouts are generated asynchronously with vLLM. Each optimization step uses 128 prompts with eight sampled trajectories per prompt, yielding 1,024 trajectories. Table 10 provides the optimization, rollout, and sequence-length settings.

Training reward. The RL reward combines final-state correctness with penalties for tool-use errors:

$$r = \begin{cases} 0, & \text{if a duplicate tool call occurs,} \\ \max\big(0,\, \mathbf{1}[D_{\text{final}} = D_{\text{gt}}] - \frac{1}{2}(m + p + i)\big)\,, & \text{otherwise.} \end{cases}$$

Here, D_{final} is the final environment state and D_{gt} is the target state. The quantities m, p, and i denote the fractions of tool calls with missing, unparsable, and invalid parameters, respectively.

C.3 Evaluation Protocol

We perform three independent evaluation runs on each benchmark subset, using the complete task set in every run. Thinking is enabled throughout, and GPT-4.1 serves as the user simulator. The agent and simulator prompts are provided in Appendix D.

Table 10: Reinforcement learning configuration. The reference model is initialized from the same checkpoint as the actor, although no KL regularization term is used.

Setting	Value
Policy optimization	GSPO with group-relative advantages and sequence-level loss
Rollout engine	Asynchronous vLLM; tensor parallelism of 2
Prompts per step	128
Rollouts per prompt	8
Trajectories per step	1,024
Maximum prompt length	4,096 tokens
Maximum response length	30,000 tokens
Maximum tool-result length	2,048 tokens
Maximum assistant turns	50
Sampling temperature	0.7
Top-p / top-k	1.0 / disabled (-1)
Learning rate	10^{-6}
Warmup steps	20
Weight decay	0.1
Gradient clipping threshold	1.0
Lower / upper clipping thresholds	0.0003 / 0.0004
Entropy coefficient	0
KL regularization	None

τ-bench and τ^2-Bench. We enable interleaved reasoning and set the assistant and user-simulator generation limits to 10,000 and 8,192 tokens, respectively. User-simulator requests have a timeout of 600 seconds. The agent temperature is 0.7, except for the non-solo Telecom evaluation, which uses 0.0. We do not override the implementation's default top-p setting. Interaction budgets are given in Table 8. When a context-length error occurs, generation is retried with an adaptively reduced token budget. If the available budget falls below 512 tokens, the task terminates with an error.

VitaBench. Each run includes one trial per task, using Chinese prompts and a configured seed of 300. GPT-4.1 serves as both the user simulator and the judge in the trajectory evaluator. A trajectory terminates when it reaches 300 interaction steps, accumulates 10 consecutive tool errors, or produces an invalid agent message. Such premature terminations receive a score of zero. A task is considered successful only when its final reward equals 1.0.

Failure handling and run completeness. We use fixed-denominator scoring: every task in the evaluated subset contributes to the success-rate denominator. Task-level exceptions, invalid outputs or tool calls, timeouts, and premature termination are recorded as failures with a score of zero. A run is complete only when it contains an outcome record for every expected task. Runs with missing task records are considered incomplete and are excluded from aggregation; recorded task failures remain in the denominator.

Metrics and aggregation. For each complete run, we compute the task success rate as the percentage of tasks successfully completed according to the corresponding benchmark evaluator. All tasks in the evaluated subset are included in the denominator, with failures assigned a score of zero. We report the mean success rate across three independent runs.

D PROMPTS

We provide condensed versions of the principal synthesis and interaction prompts in the same order as the methodology in the main paper. Each block combines the core instructions, representative inputs, and output structure. Repeated context, repair boilerplate, examples, and model-specific variants are omitted; wording and schema displays are shortened. Braced placeholders are filled at runtime, and ellipses denote omitted schema entries. Deterministic assembly and graph extraction use no prompts.

D.1 SCAFFOLD SYNTHESIS

Prompt 1: Object Drafting

```
Design {num_objects} distinct, interacting object classes covering
diverse sub-areas of {domain}. Give each class an {object_name}_id
identifier and 3-5 other attributes. Avoid treating parts of one
entity as separate objects. Use realistic field names and types.
Input: {domain}, {num_objects}.
Output: one YAML code block with domain, domain_id, and objects;
each object contains id_field and fields (a mapping from field names
to types).
```

Prompt 2: Environment Grouping

```
Group the given objects into {num_envs} coherent environments within
{domain}. Each environment contains 3-5 objects that interact
naturally. Objects may appear in multiple environments, but every
object must appear at least once. Name each environment after its
scenario; derive a snake_case environment_id.
Input: {domain}, {num_envs}, {objects_list}.
Output, repeated per environment:
## Environment: Environment Name
- environment_id: env_name_snake_case
- objects: [ObjectA, ObjectB, ObjectC]
```

Prompt 3: MDG Refinement

```
Refine the object drafts into an ENV Blueprint. Mark ID links with
ref(Object) or List[ref(Object)]. Use shared(Object.field) only for
semantically aligned, non-ID attributes; it does not enforce runtime
equality. All targets must exist. Permit bidirectional ID links only
for true many-to-many relations represented by List[ref] on both
sides; otherwise keep one direction.
Input: {domain}, {environment}, {object_drafts_yaml}.
Output: one YAML code block with environment, environment_id, and
objects; each object contains id_field and fields with explicit
relation annotations.
```

D.2 Instruction Synthesis

Prompt 4: Dependency Instructions

```
Create dependency instructions from one realistic issuer
perspective. Each uses exactly two graph-connected object classes
and ends in a state change supported by an existing manipulate
target_tool. Require cross-object reasoning beyond a direct ID or
simple ID handoff. Write one high-level sentence per instruction,
without tool names or concrete IDs/dates/amounts. Supply complete
selection rules and tie-breakers for a unique target/set.
Input: {instruction_count}, {env_schema_code},
{object_tool_graph_json}.
Output: ANALYSIS: then JSON: with perspective and
dependency_instructions: [{instruction, analysis, objects: [ObjectA,
ObjectB], target_tool}, ...].
```

Prompt 5: Database Constraints

```
Infer initial-state constraints for every dependency instruction,
using only schema fields and graph relations. Require meaningful
candidate variation and distractors. Use a 3-4 candidate core pool
when applicable, unless the instruction requires a larger set. For
filters, include both qualifying and nonqualifying candidates in
that pool. Full selection rules must determine one winner or a
unique batch membership set. Do not invent concrete IDs.
Input: {env_schema_code}, {object_tool_graph_json},
{dependency_instructions_json}.
Output JSON: {list_of_constraints: [{dependency_instruction_index,
constraints: ["...", ...]}, ...]}; exactly one item per input index.
```

Prompt 6: Database Initialization

```
Generate a small init_config for ENV(**init_config), matching
container names and value types. Use at least three instances per
top-level dict, unique IDs, consistent references, and concrete
instances of every graph edge. Satisfy all dependency constraints,
including candidate variability, distractors, 3-4 core pools when
specified, and a uniquely determined winner or batch. Use YYYY-MM-DD
or YYYY-MM-DD HH:MM for dates/times.
Input: {env_schema_code}, {object_tool_graph_json},
{dependency_instructions_json}, {list_of_constraints_json}.
Output: # Analysis, then # Init Config with one JSON code block.
```

Prompt 7: Shadow Database

```
Add plausible but task-irrelevant shadow attributes only to
searchable object classes. Never repeat/modify real fields or add
IDs, join keys, relations, or business control fields. Prefer 2-4
attributes per class with simple values. Provide complete
default_attrs for future instances and varied overrides for every
existing instance; keep the overlay small.
Input: {env_schema_code}, {tool_metadata_json}, {init_config_json}.
Output: # Analysis, then # ShadowDB with JSON: {objects: {ClassName:
{default_attrs: {...}, instances: {id: {...}, ...}}}}.
```

Prompt 8: Candidate Task Paths

```
Create one coherent task with at least {min_subtasks} subtasks,
including at least one state-changing subtask. Bind each to 1-2
existing object classes and only their tools. A later write must use
  information from an earlier subtask.
Include comparison or batch selection when supported, with a
complete rule yielding a unique target/set. Preserve DI-specific
candidate constraints. Give exact non-derivable write constants;
omit tool-retrievable answers and tool names. At least one subtask
must use a valid dependency_instruction_index.
Input: {init_config_json}, {env_schema_code},
{object_tool_graph_json}, {dependency_instructions_json},
{list_of_constraints_json}.
Output: ANALYSIS: then JSON: {task_summary, subtasks: [{objects,
subtask, dependency_instruction_index (when directly using that
DI)}, ...]}.
```

Prompt 9: Ground-Truth Solver

```
Solve only the current subtask, using prior subtask history when
provided. Completion requires successfully executing every requested
operation. Call only allowed tools, with all required parameters and
valid schema types/enums. Propose at most one tool call per turn.
When complete, omit the tool call. If validation errors are
provided, fix only those errors. Use no markdown.
Input: {subtask_index}, {subtask_text}, {objects_json},
{allowed_tools_json}, {prior_messages_json}.
Output: one <think>...</think> block, optionally followed by
<tool_call>{"name": "...", "arguments": {...}}</tool_call>.
```

Prompt 10: Ground-Truth Reviewer

```
Pass only if the subtask is fully achieved, its information is
appropriate, and every decisive argument is grounded in visible
evidence. Unsupported guesses, wrong targets, material side effects,
or insufficient evidence fail. Tool errors alone do not imply
failure. ROUND1: distinguish subtask_issue from solver_issue; repair
only flawed subtask text, preserving scope and non-derivable
constants without leaking tool-retrievable answers. ROUND2: give a
final verdict without repair.
Input: {mode}, {subtask}, {allowed_tools_json},
{prior_messages_json}, {tool_steps_json}, {state_before_json},
{state_diff_json}.
Output JSON: {verdict: "pass"|"fail", reason}; on ROUND1 failure
also include classification, and refined_subtask only for
subtask_issue.
```

Prompt 11: Task Instruction Assembly

```
Rewrite verified subtasks into a natural goal-level request or
hidden user scenario. Preserve all deliverables, selectors, exact
create/update values, final states, ordering, and dependencies.
Absorb supporting lookups into selection conditions. Add no facts or
tool-returned answers. Do not mention tools, schemas, subtasks,
execution plans, or validation internals.
Input: {subtask_list}; apply {validation_errors_block} if provided.
RL output: {"task_instruction": "..."}; one cohesive user request.
SFT output: {"reason_for_call": "...", "known_info": "..."}. Write
the SFT scenario in second person. Keep reason_for_call high-level
and put all exact user-known details needed to reproduce the
verified task in known_info; use null only when no such facts exist.
```

D.3 Environment Synthesis

Prompt 12: Schema and Environment Classes

```
Translate the blueprint into Pydantic v2: one BaseModel per object
and one ENV class. Map ref to str, List[ref] to List[str], and
shared to its target field's type. Add no fields, Optional/Union
types, or None defaults. For each object, ENV contains one Dict[str,
Object] with Field(default_factory=dict), named by pluralizing the
last word of the object's snake_case name.
Input: {env_blueprint_yaml}; apply {validation_errors_block} if
provided.
Output: a single complete Python code block defining exactly these
classes.
```

Prompt 13: Base Tool Design

```
Fill exactly one tool_name and tool_logic per supplied spec_id. Use
unique snake_case names and implementable ENV operations. State
Inputs; searches also state Returns and return full objects.
Creation must validate all parent IDs before mutation, store the
object, and perform specified incoming-list cascades without
duplicates. On validation failure, mutate nothing. Deletion must
remove the object and clean all specified incoming list references.
Input: {object_definitions}, {pydantic_class_code},
{base_tool_specs_json}.
Output: JSON with environment_name, environment_id, and base_tools:
[{spec_id, tool_name, tool_logic}, ...].
```

Prompt 14: Relation Tool Design

```
Fill every provided edge-tool spec exactly once; invent no
additional tools. Use unique names absent from the existing list.
Inputs must exactly match required_inputs. Searches return full
objects of expected_returns. ref_update validates the target before
mutation. shared_search returns matching source objects;
shared_update changes only the source field, without propagation.
Input: {pydantic_class_code}, {edges_json},
{existing_tool_names_json}, {edge_tool_specs_json}.
Output: JSON with environment_name, environment_id, and
edge_tool_specs: [{spec_id, tool_name, tool_logic}, ...].
```

Prompt 15: Additional Tool Design

```
Design realistic workflow tools, including both search and
manipulate tools. Use unique snake_case names, valid object classes,
and implementable logic. Do not duplicate existing/reserved
capabilities. Searches are read-only, declare Inputs and Returns,
and return full candidate objects. Leave ranking and final decisions
to the agent; avoid find_best/choose shortcuts and backend-admin,
maintenance, or data-repair manipulation tools.
Input: {object_definitions}, {pydantic_class_code},
{existing_tool_names_json}, {reserved_capabilities_json},
{relation_fields_json}.
Output: JSON with environment_name, environment_id, and other_tools:
[{object, tool_name, tool_type, tool_logic}, ...].
```

Prompt 16: Tool Filtering

```
Infer the reasonable operator role from the environment, schema,
relations, and tool semantics. Keep bounded workflow actions within
that role. Filter tools requiring outside authority: administration,
unrestricted repair, master-data governance, or broad destructive
operations. Usually filter raw create/delete wrappers unless normal
for that role. Keep ambiguous but plausible bounded workflow
actions. Do not output the inferred role.
Input: {environment_name}, {object_definitions},
{pydantic_class_code}, {manipulation_tools_json}.
Output JSON only: {"filtered_out_tools": ["rejected_tool_name",
...]}.
```

Prompt 17: Tool Documentation

```
Write a concise English docstring describing business intent and
behavior. Include an Args: section covering every parameter. Do not
include Returns:, Errors:, Side Effects:, or Notes: sections. If
time_format_required is true, include exactly: Time format:
YYYY-MM-DD or YYYY-MM-DD HH:MM
Input: {tool_name}, {signature_block}, {tool_logic},
{constraints_block}.
Output: raw docstring text only; no header, triple quotes, or code
fences.
```

Prompt 18: Tool Implementation

```
Generate one ENV method with exactly the supplied signature,
parameter order, and types. Assume schema-valid inputs; check
business preconditions and record existence. Catch errors instead of
raising exceptions. Searches must not mutate state. Return success
plus JSON-serializable data for searches, or a message for
manipulations; on failure return success=false and error. Include a
concise English docstring with Args and the required time format.
Input: {env_code}, {tools_quick_ref}, {signature_block},
{tool_logic}, {tool_type}, {tool_kind}, {time_format_required}.
Output: # Analysis, then # Code with one Python method definition.
```

Prompt 19: Test State Generation

```
Generate a JSON init_config usable directly as ENV(**init_config).
Match all ENV field names and types. Include multiple entities in
each top-level dict (3-5 recommended), with unique IDs and
consistent cross-entity references. Use fictitious, realistic data
and varied attribute values. Format dates as YYYY-MM-DD or
YYYY-MM-DD HH:MM. Correct any supplied previous error.
Input: {blueprint_yaml}, {env_schema_code}, {previous_error}.
Output: the initialization JSON only; no markdown or commentary.
```

Prompt 20: Test Call Generation

```
Choose one exploratory test call from eligible_tools, respecting the
per-tool cap. If uncovered_positive_tools is nonempty, select one
for a positive test; otherwise prioritize coverage and approach
negative_ratio. Arguments must always satisfy the tool schema.
Positive tests use coherent existing data; negative tests use
schema-valid nonexistent IDs or semantic boundary cases.
Input: {state_summary_json}, {tools_quick_reference},
{test_summary}, {eligible_tools_json}, {uncovered_positive_tools},
{negative_ratio}, {per_tool_max_calls}, {nonexistent_id_hint}.
Output JSON: {thought, selected_tool, selected_case_type,
arguments}, where selected_case_type is "positive" or "negative".
```

Prompt 21: Execution Checking

```
Judge the executed call against its specification and observed state
change. A positive call should succeed if preconditions hold. A
negative call may correctly return success=false and still Pass if
state remains safe. Assign Warning for robustness/validation gaps,
or Fail for incorrect behavior, unsafe mutation, rule violations, or
major logic mismatch.
Input: {case_type}, {tool_schema_json}, {tool_description},
{method_source}, {arguments_json}, {observation_json},
{state_before_json}, {state_after_json}, {state_diff_json}.
Output JSON: {result: "Pass"|"Warning"|"Fail", analysis,
error_reason}; error_reason is "No error" for Pass, otherwise the
cause and suggested fix.
```

D.4 Interaction and Training

Prompt 22: Conversation Action Agent (SFT)

```
Fulfill the user's request using available tools and environment
feedback. Supply complete, valid arguments; query relevant
information before updates and read each result before choosing the
next action. Multiple calls per turn are allowed, with at most five
calls in one assistant turn. Ask concise questions only for necessary
information that tools cannot retrieve. Do not expose hidden
environment details.
Input: {environment_id}, {environment_name}, user messages, tool
schemas, and successive tool results.
Communicate naturally with the user. The user simulator ends the
conversation with ###STOP###; never emit that token yourself.
```

Prompt 23: User Simulator (SFT Conversation)

```
Play a realistic user following the hidden scenario {task}, which
contains Reason for call and Known info. Produce one natural reply
at a time. Introduce each goal at a high level and disclose concrete
details progressively when asked or needed to proceed. Avoid dumping
the complete parameter bundle, including when introducing later
subtasks. Preserve exact known values; never invent missing
information. State business goals rather than tool-use plans. If a
requested detail is unknown, say so.
Input: {task} and the ongoing conversation with the action agent.
Output: only the user's next plain-text message; once the goal is
satisfied, output exactly ###STOP### as a standalone message.
```

Prompt 24: Tool-Use Agent (RL Rollout)

```
Solve the task by interacting with the provided environment. Use only
the available tools and ground every argument in the task or a prior
observation. Query the environment when an identifier or value must be
derived; do not invent records or parameters. Read each tool result
before choosing the next action, satisfy every requested state change,
and stop when the target state is reached or no valid action remains.
Input: {environment_id}, {environment_name}, {prompt}, serialized tool
schemas, and successive tool observations.
Output: tool calls in the configured Hermes multi-turn format. A
natural-language response must not replace a required tool call.
```

E END-TO-END TASK EXAMPLE

We conclude with a representative accepted artifact from the *Service Quality Control* environment. Following the layered example presentation of Song et al. (2026), we separate the task, its task-relevant initial state, the verification targets, and the reference interaction. The task contains seven linked subtasks. Its accepted reference path uses 18 tool calls from 12 distinct tools and produces five final-state effects.

E.1 ENVIRONMENT RELATIONS AND TASK

The executable environment contains six object classes. The relations below are the portions of the MDG exercised by this task; identifier-valued fields are preserved by the executable schema and tool implementations.

Example 1: Task-Relevant MDG

```
Customer.booking_ids  -> Booking.booking_id
Booking.service_id    -> Service.service_id
Booking.provider_id   -> ServiceProvider.provider_id
Review.booking_id     -> Booking.booking_id
Review.customer_id    -> Customer.customer_id
Review.provider_id    -> ServiceProvider.provider_id
Dispute.booking_id    -> Booking.booking_id
```

Example 2: Agent-Facing Task

```
For customer Maria (C651065), upgrade her confirmed booking to the
most expensive electrical service whose base price is no greater than
the average base price of her past services. For James O'Brien's
plumbing booking B262014, assign the verified provider with the highest
rating in the same service category as its current service. For Aisha
Patel (C143334), submit a review for completed booking B603311 with
provider P987812, using her most common past rating (choose the higher
rating in a tie), comment "Satisfactory service", and date 2024-09-15.
For booking B867365, escalate the open dispute with the smallest dispute
ID (the task's ordering rule). Finally, register
provider P999999 as "New Plumbing Co." in the plumbing category, with a
base rate equal to the average base price of all plumbing services,
rating 0.0, and verification status "pending".
```

The instruction provides constants required for writes but withholds the records and values that can be derived through tools. In particular, the new service, replacement provider, review rating, target dispute, and provider base rate must all be inferred from the initial state.

E.2 TASK-RELEVANT INITIAL STATE

The following compact view contains every candidate needed by the accepted reference path. Attributes unrelated to the task are omitted for readability; the executable environment retains the complete records.

Example 3: Initial State

```
customers:
  C651065: {name: Maria Garcia,
            booking_ids: [B836293, B169961, B146044]}

bookings:
  B836293: {service_id: S401246, status: completed}
  B169961: {service_id: S210898, status: confirmed}
  B146044: {service_id: S049406, status: completed}
  B262014: {service_id: S574527, provider_id: P336475,
            status: requested}
  B603311: {service_id: S023825, provider_id: P987812,
            status: completed}
  B867365: {service_id: S810235, provider_id: P543842,
            status: completed}

electrical_services (service_id: base_price):
  S049406: 100.0, S210898: 120.0,
  S091501: 140.0, S401246: 200.0
plumbing_services (service_id: base_price):
  S810235: 100.0, S574527: 150.0,
  S166600: 200.0, S224476: 250.0

plumbing_providers (provider_id: rating, verification):
  P336475: 3.5 verified, P657647: 3.8 verified,
  P901777: 4.2 verified, P264438: 4.5 verified,
  P543842: 4.0 pending

reviews for C143334 (review_id: rating):
  R081109: 4, R091369: 4, R264646: 5
open disputes for B867365:
  D732673, D878974
```

The completed services in Maria's history cost 200 and 100, so their mean is 150. Among the qualifying electrical services, `S091501` is the most expensive at 140. The highest-rated verified plumbing provider is `P264438` at 4.5. Aisha's rating multiset is $\{4, 4, 5\}$, whose mode is 4. The plumbing-service mean is $(100 + 150 + 200 + 250)/4 = 175$.

E.3 Solution Derivation

The reference solution is obtained by following the dependency edges exposed by the task, rather than by reading any of the hidden write arguments directly. Table 11 summarizes the intermediate evidence and the resulting decision for each subtask. The first subtask joins a customer to her bookings, separates completed history from the confirmed target, aggregates service prices, and then applies a category and price constraint before selecting the maximum. The provider subtask composes a booking-to-service lookup with a verification filter and a rating argmax. The review subtask derives a mode, while the dispute subtask uses the task-defined ID ordering after filtering by booking and status. The final provider creation uses a second aggregation over a complete category slice.

This derivation exposes why the example is a multi-step dependency task. The agent must traverse identifier-valued relations, retain only records satisfying semantic predicates, compute aggregates over retrieved records, and carry the result into later state-changing calls. A locally plausible action is insufficient: the final checker compares all five effects jointly, including the newly created records and both updated references.

Table 11: Intermediate solution derivation for the complex task. Every derived write argument is supported by an earlier observation from the same executable environment.

Subtask	Evidence and reasoning	Derived write
Maria's upgrade	Completed prices are 200 and 100, giving limit 150. Electrical services at or below the limit are 100, 120, and 140; choose the maximum, `S091501`.	`B169961.service_id := S091501`
Provider reassignment	`B262014` points to `S574527`, whose category is plumbing. Among plumbing providers, retain verified records and choose the highest rating, 4.5.	`B262014.provider_id := P264438`
Review submission	Aisha's observed ratings are $[4, 4, 5]$; the mode is 4. The completed booking and provider identifiers are supplied by the task and checked against the environment.	Add `R999999` with rating 4
Dispute escalation	The open disputes for `B867365` are `D732673` and `D878974`. The task's ID-order rule selects the smaller ID.	`D732673.status := escalated`
Provider creation	The plumbing price slice is $[100, 150, 200, 250]$; its mean is 175. The remaining provider fields are fixed by the task.	Add `P999999` with base rate 175

Table 12: Verification targets for the complex task example. Each row is checked against the final executable state rather than the agent's textual completion.

No.	Final-state condition
1	Booking `B169961` references service `S091501`.
2	Booking `B262014` references provider `P264438`.
3	Review `R999999` exists for `B603311` and `P987812`, with rating 4, the specified comment, and date.
4	Dispute `D732673`, the first open dispute under the task's ordering rule, has status `escalated`.
5	Provider `P999999` exists with category `plumbing`, base rate 175.0, rating 0.0, and status `pending`.

E.4 Reference Interaction and Final State

The accepted reference interaction is shown below. Tool responses are reduced to the fields used by the next decision, while all calls and derived write arguments are retained.

Example 4: Accepted Reference Tool Path

```
 1 fetch_customer_profile(C651065)
   -> booking_ids [B836293, B169961, B146044]
 2 fetch_booking_record(B836293) -> S401246, completed
 3 fetch_booking_record(B169961) -> S210898, confirmed
 4 fetch_booking_record(B146044) -> S049406, completed
 5 fetch_service_details(S401246) -> electrical, price 200.0
 6 fetch_service_details(S049406) -> electrical, price 100.0
 7 find_services_by_category(electrical)
   -> prices {100.0, 120.0, 140.0, 200.0}; select S091501
 8 set_booking_service(B169961, S091501) -> success

 9 fetch_booking_record(B262014) -> service S574527
10 fetch_service_details(S574527) -> category plumbing
11 find_providers_by_category(plumbing)
   -> highest-rated verified provider P264438 (4.5)
12 set_booking_provider(B262014, P264438) -> success

13 search_reviews_by_customer_and_rating(C143334, min_rating=1)
   -> ratings [4, 4, 5]; mode 4
14 submit_service_review(R999999, B603311, C143334, P987812,
   rating=4, comment="Satisfactory service", date=2024-09-15)
   -> success

15 search_disputes_by_booking_and_status(B867365, open)
```

```
   -> [D732673, D878974]; select D732673
16 escalate_dispute_case(D732673) -> success

17 find_services_by_category(plumbing)
   -> prices [100.0, 150.0, 200.0, 250.0]; mean 175.0
18 register_provider_account(P999999, "New Plumbing Co.", plumbing,
   base_rate=175.0, rating=0.0, verification=pending) -> success
```

Example 5: Exact Final-State Difference


```
{
  "bookings": {
    "B169961": {"service_id": {"changed": {
      "old": "S210898", "new": "S091501"}}},
    "B262014": {"provider_id": {"changed": {
      "old": "P336475", "new": "P264438"}}}
  },
  "reviews": {
    "R999999": {"added": {"review_id": "R999999",
      "booking_id": "B603311",
      "customer_id": "C143334", "provider_id": "P987812",
      "review_rating": 4, "review_comment": "Satisfactory service",
      "review_date": "2024-09-15"}}
  },
  "disputes": {
    "D732673": {"dispute_status": {"changed": {
      "old": "open", "new": "escalated"}}}
  },
  "service_providers": {
    "P999999": {"added": {"provider_id": "P999999",
      "provider_name": "New Plumbing Co.",
      "business_name": "New Plumbing Co.",
      "service_category": "plumbing", "base_rate": 175.0,
      "provider_rating": 0.0, "verification_status": "pending"}}
  }
}
```


This example illustrates three forms of dependency in one task: values derived by aggregation (means and a mode), targets selected by constrained ranking, and later writes grounded in identifiers returned by earlier queries. The final-state comparison verifies all five effects jointly, so a trajectory that completes only a subset does not receive a successful task outcome.