BEHAVIORPILOT: BEHAVIOR-GUIDED DATA SYNTHESIS FOR TOOL-USING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Training tool-using agents requires diverse interaction data and reliable execution feedback. Recent approaches adopt *scenario-centric synthesis*, generating environments and tasks across domains and tool configurations. However, scenario diversity does not ensure broad behavioral coverage: tasks from different scenarios may repeatedly exercise the same interaction patterns while leaving others underrepresented. To enable explicit control over behavioral coverage, we introduce *tool-use behaviors* as a representation of recurring interaction patterns and propose **BehaviorPilot**, a behavior-guided data synthesis framework. BehaviorPilot integrates three modules: *behavior taxonomy induction* to build a hierarchical behavioral representation from open-source trajectories for task synthesis and model diagnosis; *behavior-conditioned data synthesis* to generate executable tasks targeting selected behavior combinations and retain rollouts that pass task verification and cover the target behaviors; and *iterative behavior-guided refinement* to target model weaknesses by allocating additional synthesis budget in proportion to per-behavior failure rates on a fixed, behavior-balanced validation set. Applied to Qwen3-8B, BehaviorPilot achieves the best benchmark-average scores among the evaluated baselines on VitaBench, BFCL V4 multi-turn, and τ^2-Bench. Notably, the resulting 8B model surpasses Qwen3-32B in overall performance.

1 INTRODUCTION

Large language model (LLM) agents interact with external systems through tools to retrieve information, execute operations, and modify environment states (Yao et al., 2022; Schick et al., 2023). Effective tool use requires more than selecting a function: agents must construct valid arguments, propagate information across calls, satisfy constraints, and adapt to intermediate observations (Li et al., 2023; Lu et al., 2025; Yao et al., 2025). We propose to explicitly represent these recurring interaction patterns as *tool-use behaviors*. Learning to apply and combine these behaviors is central to developing reliable tool-using agents (Yu et al., 2025; Zhang et al., 2026).

Agents learn these interaction behaviors through supervised fine-tuning (SFT) on tool-use trajectories (Qin et al., 2024; Zeng et al., 2024) and reinforcement learning (RL) with task-level feedback (Xi et al., 2025; Wang et al., 2026). The effectiveness of both approaches depends on the quality and diversity of the interaction experiences represented in training data, as well as the reliability of task feedback (Liu et al., 2024; Prabhakar et al., 2025). To provide such experiences at scale, a common approach is *scenario-centric synthesis*, which constructs scenarios across different domains and tool configurations (Song et al., 2026; Hu et al., 2025). Within each scenario, tasks are generated from the available tools, data, and constraints, producing training experiences shaped by the selected environments and task designs (Dong et al., 2026).

However, scenario-centric task synthesis does not necessarily ensure broad coverage of tool-use behaviors. As illustrated in Figure 1, tasks from different domains may still share the same interaction pattern: retrieving a passenger identifier before querying a booking and retrieving a customer identifier before querying an order both involve cross-step argument binding. Conversely, a single environment can support distinct behaviors, including constrained retrieval, and error recovery. Scenario diversity alone therefore does not fully characterize the behavioral composition of training data. Without an explicit behavioral representation, it is difficult to assess which patterns the data covers and determine what additional tasks to generate when a model remains weak.

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.

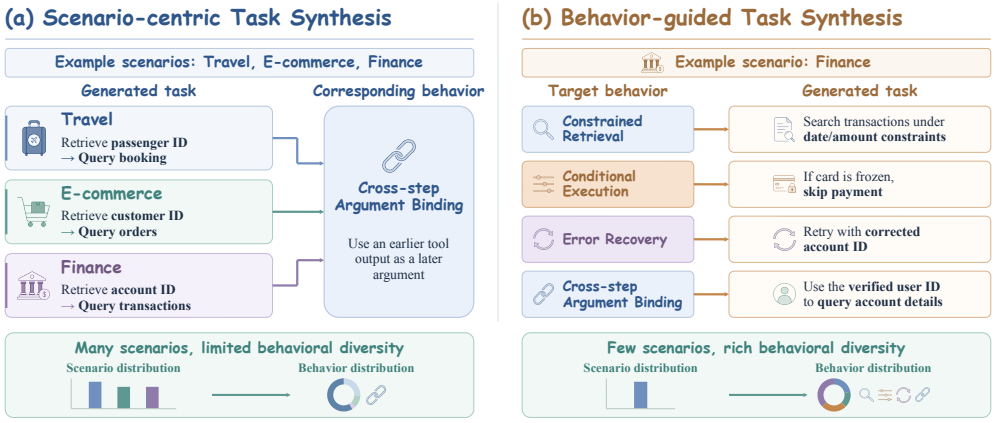


Figure 1: Scenario-centric versus behavior-guided task synthesis. Tasks from different scenarios may share the same behavior, while a single scenario can support tasks targeting distinct behaviors.

To make behavioral coverage explicit and target model weaknesses, we propose **BehaviorPilot**, a behavior-guided data synthesis framework for tool-using agents. BehaviorPilot first induces a hierarchical taxonomy of recurring tool-use behaviors from open-source trajectories, providing a shared representation for both task synthesis and model diagnosis. It then generates behavior-conditioned executable tasks and retains rollouts that pass task verification and cover the target behaviors. The verified trajectories support supervised fine-tuning, while the executable tasks and verifiers support reinforcement learning. During iterative refinement, BehaviorPilot allocates additional synthesis budget in proportion to per-behavior failure rates measured on a fixed, behavior-balanced validation set. Applied to Qwen3-8B, BehaviorPilot outperforms all evaluated baselines on the average scores of VitaBench, BFCL V4 multi-turn, and τ^2-Bench. It achieves an overall score of 40.6, exceeding Qwen3-8B and Qwen3-32B by 12.8 and 4.7 points, respectively.

Our contributions are as follows:

- We introduce *tool-use behaviors* as explicit targets for data synthesis and induce a hierarchical taxonomy from tool-use trajectories, addressing the limited control over behavioral coverage in scenario-centric synthesis.
- We propose **BehaviorPilot**, which integrates behavior taxonomy induction, behavior-conditioned data synthesis, and iterative behavior-guided refinement to translate behavioral targets into executable tasks and refine training data based on per-behavior validation.
- Extensive experiments on VitaBench, BFCL V4 multi-turn, and τ^2-Bench show that BehaviorPilot, applied to Qwen3-8B, achieves the best benchmark-average scores among the evaluated baselines and outperforms Qwen3-32B overall.

2 Related Work

2.1 Tool-Use Data Synthesis for Agent Learning

Training tool-using agents requires data connecting user requests, tool calls, and observations. Existing methods synthesize tasks and collect trajectories at scale using different execution and feedback sources. We group these methods into three approaches: real-tool-based synthesis (Qin et al., 2024; Xu et al., 2025), LLM-simulated synthesis (Liu et al., 2025; Xu et al., 2026b), and synthetic-environment-based synthesis (Song et al., 2026; Tu et al., 2026).

Real-tool-based methods construct tasks and collect trajectories using existing tools and services (Patil et al., 2024; Prabhakar et al., 2025; Liu et al., 2024). ToolLLM (Qin et al., 2024) generates instructions over real-world APIs and searches for solution paths, while TOUCAN (Xu et al., 2025) collects validated trajectories through tool execution in MCP environments. However, paid access, request limits, and tool unavailability make scaling costly and slow.

LLM-simulated methods use language models to generate interactions without executing the underlying tools. ToolACE (Liu et al., 2025) synthesizes tool-use dialogues with simulated responses, while EnvACE (Xu et al., 2026b) uses a shared policy to alternate between taking actions and rehearsing environment responses. These methods reduce dependence on external services, but simulated responses may not reflect actual tool behavior or state changes.

Synthetic-environment-based methods construct executable environments with explicit tool interfaces, persistent states such as databases, and task-specific verifiers (Song et al., 2026; Tu et al., 2026; Wang et al., 2026; Xu et al., 2026a; Dong et al., 2026). Tool calls execute code to read or update the environment state and return observations, while verifiers check whether task requirements are satisfied. This supports trajectory collection and outcome evaluation without relying on live services or LLM-simulated tool responses. BehaviorPilot follows this approach, using explicit behavioral targets to guide task generation within these environments.

2.2 Task Generation in Executable Environments

Building on executable environments, recent work studies how to scale environment construction and generate tasks that support agent learning.

Several methods emphasize scalable environment and task construction. EnvScaler (Song et al., 2026) generates environment skeletons and task scenarios with rule-based validators. ScaleEnv (Tu et al., 2026) combines procedural testing with tool-dependency graph expansion to construct executable environments and verifiable tasks. AWM (Wang et al., 2026) generates database-backed environments with executable tools and task-specific success criteria. Agent-World (Dong et al., 2026) broadens domain coverage through environment discovery and generates verifiable tasks within the resulting environments. Agent diagnosis then guides further environment and task expansion. EnvFactory (Xu et al., 2026a) grounds environment construction in online resources and guides trajectory generation with tool-interaction graphs.

Other work couples environment synthesis with multi-stage or adaptive agent training. AgentScaler (Fang et al., 2026) constructs heterogeneous executable environments and trains agents in two stages, progressing from general tool use to domain specialization. SPADE (Liu et al., 2026) jointly trains an environment designer and a reasoning agent through self-play, using hint-based regret to adapt the difficulty of generated environments.

These methods generate more varied environments and tasks or adjust task difficulty based on agent performance. However, different tasks can still exercise the same tool-use patterns, while some behaviors receive little training. In contrast, BehaviorPilot uses behavioral targets to guide generation, enabling explicit control over the types and distribution of behaviors in the generated tasks.

3 Methods

3.1 Problem Formulation

An executable environment $\mathcal{E}$ provides an underlying data state and a set of callable tools. Tool calls operate on this state and return observations according to predefined execution logic. Let $\mathcal{B} = \{b_1, \ldots, b_K\}$ denote a set of behavior groups, each describing a recurring tool-use pattern. We formulate behavior-guided data synthesis as constructing a task in $\mathcal{E}$ conditioned on a target behavior combination $\mathbf{b} \subseteq \mathcal{B}$. Each task is defined by an initial state s_0, a user request u, and a task-specific verifier. Interacting with the environment induces a trajectory

$$\tau = (u, a_1, o_1, \ldots, a_T, o_T), \tag{1}$$

where a_t denotes a tool call and o_t denotes the resulting observation. Each call may query or modify the environment state, and the verifier determines whether the interaction satisfies the task requirements. Verified trajectories provide supervised interaction data, while the corresponding tasks, environments, and verifiers support reinforcement learning.

3.2 Framework Overview

We propose **BehaviorPilot**, a behavior-guided data synthesis framework that addresses the limited control over behavioral coverage in scenario-centric synthesis. As shown in Figure 2, BehaviorPilot

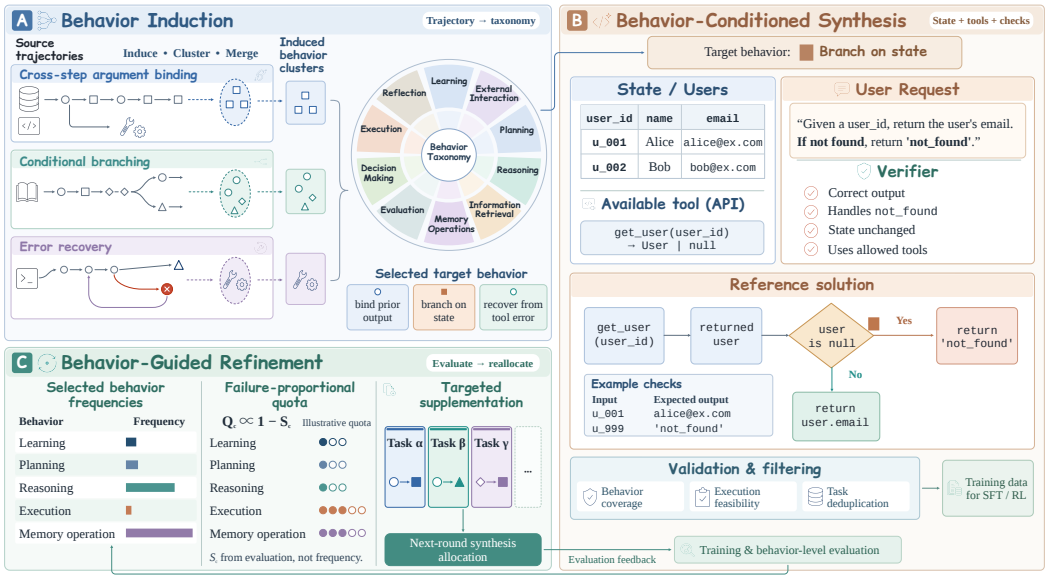


Figure 2: Overview of BehaviorPilot: behavior taxonomy induction, behavior-conditioned data synthesis, and iterative behavior-guided refinement.

first induces a hierarchical behavior taxonomy from tool-use trajectories, establishing a shared representation for task synthesis and model diagnosis. It then generates executable tasks conditioned on behavior combinations and validates task feasibility and behavioral coverage. Agent rollouts that pass task verification and cover the target behaviors are retained for SFT, while the executable tasks and verifiers support RL. During refinement, behavior-level evaluation identifies model weaknesses and directs additional synthesis through failure-rate-based budget allocation.

3.3 Behavior Taxonomy Induction

To establish a shared behavioral representation for data synthesis and model diagnosis, we construct a behavior taxonomy from publicly available tool-use trajectories. For each trajectory τ_i, an LLM examines its tool calls, arguments, and observations to produce descriptions of recurring interaction patterns, such as cross-step argument binding, conditional branching, and error recovery. These descriptions capture tool-use behaviors rather than domain-specific entities or tool names. The induction and consolidation process is summarized as

$$\mathcal{T}_{\text{src}} \xrightarrow{\text{Induct}} \mathcal{Y} \xrightarrow{\text{Cluster}} \{C_1, \ldots, C_K\} \xrightarrow{\text{Merge}} \mathcal{B} = \{b_1, \ldots, b_K\}, \tag{2}$$

where $\mathcal{T}_{\text{src}}$ is the collection of source trajectories, $\mathcal{Y}$ is the pool of induced behavior descriptions, each C_k clusters semantically similar descriptions, and b_k is the resulting Behavior Group.

We organize $\mathcal{B}$ into a three-level Category–Subcategory–Behavior hierarchy. Categories represent broad types of tool-use interactions, subcategories group related behaviors, and leaf-level behaviors specify concrete interaction patterns. Each behavior retains an operational description and representative source trajectories for task synthesis and model diagnosis. Figure 3 illustrates this hierarchy.

3.4 Behavior-Conditioned Data Synthesis

Building on the taxonomy, we use behavior combinations as explicit targets for task synthesis. To support their instantiation, we construct executable environments from LLM-generated business specifications, which guide the generation of database schemas, executable tools, and initial data, as illustrated in Figure 4. Within each environment, an LLM-based planner selects a combination $\mathbf{b} \subseteq \mathcal{B}$ supported by its tools and state transitions. It then designs a task whose goals, constraints, and tool dependencies are intended to elicit these behaviors. The environment provides the business context and executable operations, while the selected behaviors guide what the task requires the agent to do.

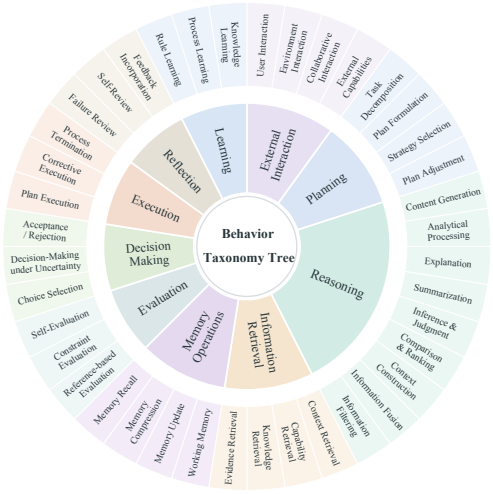


Figure 3: The hierarchical organization of the behavior taxonomy.

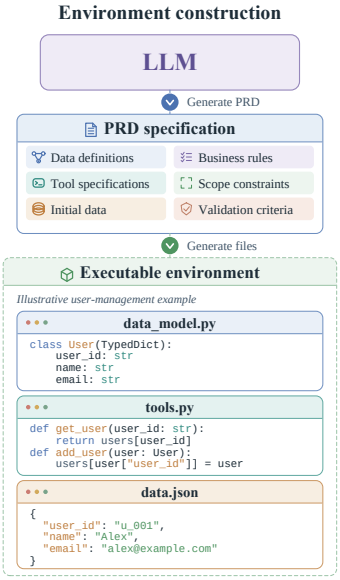


Figure 4: The workflow for generating executable environments.

The task specification is instantiated as a user request, a reference solution, and a task-specific verifier:

$$(u, r, v) = \text{Generate}(\mathcal{E}, \mathbf{b}, q), \tag{3}$$

where q denotes the planned task specification, and u, r, and v denote the user request, reference solution, and verifier, respectively. To verify behavioral coverage, an LLM identifies the behaviors involved in the reference solution and checks whether they include all target behaviors in $\mathbf{b}$. Formally, the behavioral coverage check requires

$$\mathbf{b} \subseteq \widehat{\mathcal{B}}(r), \qquad \widehat{\mathcal{B}}(r) = \text{Identify}(r), \tag{4}$$

where $\widehat{\mathcal{B}}(r)$ is the set of behaviors identified in the reference solution. If any target behavior is missing, we revise the task and reference solution and repeat the check. After the behavioral coverage check, we remove duplicate or highly similar tasks and execute their reference solutions to confirm feasibility. We then collect agent rollouts for the validated tasks. An LLM identifies the behaviors in each rollout and checks whether it covers all target behaviors, without requiring the same tool-call sequence as the reference solution. Only rollouts that pass both the task verifier and this trajectory-level behavior-coverage check are retained for SFT. The validated tasks can also be used for RL, with their verifiers providing reward feedback during agent–environment interaction.

3.5 Iterative Behavior-Guided Refinement

We further refine the training data by identifying behaviors on which the model remains weak and targeting them in subsequent synthesis rounds. We first construct a fixed, behavior-balanced validation set using the same executable environments as training but a separate set of tasks. Each validation task has one designated target behavior, with an equal number of tasks assigned to each behavior group, enabling consistent measurement of behavior-level success rates.

After each training round t, we evaluate the updated model on this validation set. For each behavior b_k, the success rate is computed as

$$s_k^{(t)} = \frac{1}{N_k} \sum_{i=1}^{N_k} \mathbf{1}\left[v_{ki}(\tau_{ki}^{(t)}) = 1\right]. \tag{5}$$

where N_k is the number of validation tasks targeting b_k, $\tau_{ki}^{(t)}$ is the rollout produced on the i-th such task, and v_{ki} is its task verifier. We allocate a fixed synthesis budget across behaviors in proportion

to their failure rates:

$$Q_k^{(t+1)} = B_t \cdot \frac{1 - s_k^{(t)}}{\sum_{j=1}^{K}(1 - s_j^{(t)})}, \tag{6}$$

where B_t is the total synthesis budget for the next round and $Q_k^{(t+1)}$ is the number of new training tasks allocated to behavior b_k. Behaviors with lower success rates receive more tasks. Each generated task counts toward one primary behavior's quota, even when it involves multiple behaviors. The synthesis pipeline receives only aggregate behavior-level statistics, not validation tasks or solutions, limiting direct adaptation to the validation set.

We repeat evaluation, targeted synthesis, and training until predefined behavior-level targets or budget limits are reached. Public benchmarks are reserved for final generalization evaluation and do not guide synthesis or stopping decisions.

4 Experiments

4.1 Datasets and Evaluation Metrics

Following prior work on training tool-using agents (Song et al., 2026; Tu et al., 2026; Wang et al., 2026), we evaluate on three public agentic tool-use benchmarks. **BFCL V4** (Patil et al., 2025) evaluates function-calling capabilities across diverse tool schemas and parameter configurations. We use its *Multi-Turn* subset, which tests whether agents can maintain context, select appropriate tools across turns, and handle cases where required functions or parameters are not explicitly provided. It consists of Base, Long Context, Missing Function, and Missing Parameter tasks. **τ^2-Bench** (Barres et al., 2025) evaluates conversational agents in stateful environments in which tool calls can query and update persistent records. We report task-level success on its Retail, Telecom, and Airline domains. **VitaBench** (He et al., 2026) covers multi-step and multi-turn tool use in life-service scenarios, requiring agents to integrate information across tools and satisfy user goals within realistic business workflows. We report task-completion results on its Cross, Delivery, In-store, and OTA subsets. Results are reported as task success rates averaged over four evaluation runs.

4.2 Implementation Details

For behavior taxonomy induction, we use DeepSeek-V4-Pro-Preview (DeepSeek-AI, 2026) to extract and consolidate behavior descriptions from source trajectories. The resulting taxonomy comprises 10 categories, 40 subcategories, and 100 leaf-level behaviors.

For behavior-conditioned data synthesis, we also use DeepSeek-V4-Pro-Preview to construct 100 executable environments and generate task–solution–verifier triples conditioned on selected behavior combinations. Qwen3.6-27B (Qwen Team, 2026) serves as the teacher model for trajectory collection, yielding more than 9,000 verified trajectories for SFT.

We adopt Qwen3-8B (Qwen Team, 2025) as the base model and train it with SFT using MS-SWIFT (Zhao et al., 2024), followed by agentic RL using ROLL (Wang et al., 2025). RL employs GRPO (Shao et al., 2024) with rewards provided by task verifiers. Both stages use the same synthesized environments, while behavior-level validation guides additional data synthesis during iterative refinement. For computational efficiency, we use SFT only to evaluate model performance in the ablation and analysis experiments.

4.3 Main Results

We compare BehaviorPilot with Qwen3 models (Qwen Team, 2025) at 4B, 8B, and 32B scales, as well as representative tool-use post-trained models, including TOUCAN (Xu et al., 2025), Simulator (Li et al., 2025), ScaleEnv (Tu et al., 2026), AWM (Wang et al., 2026), EnvFactory (Xu et al., 2026a), EnvScaler (Song et al., 2026), EnvACE (Xu et al., 2026b), and SPADE (Liu et al., 2026). Table 1 shows that BehaviorPilot achieves the best average score on all three benchmarks, reaching 20.5 on VitaBench, 53.3 on BFCL V4 Multi-Turn, and 48.1 on τ^2-Bench. With an overall score of 40.6, BehaviorPilot ranks first among all evaluated methods. Its consistent lead across the three

Table 1: Main results on agentic tool-use benchmarks. Within the Tool-Use Post-Trained Models group, **bold** and underlined values indicate the best and second-best results, respectively.

Method	VitaBench					BFCL V4 MT					τ^2-Bench				Overall
	Cross	Delivery	In-store	OTA	**Avg.**	Base	Long Ctx	Miss Func	Miss Param	**Avg.**	Retail	Telecom	Airline	**Avg.**	
Base Models (4B–32B)															
Qwen3-4B	1.8	13.7	18.2	4.8	9.6	51.0	39.0	34.0	28.0	38.0	32.7	13.2	38.0	27.9	25.2
Qwen3-8B	1.2	15.5	24.8	4.0	11.4	55.5	41.0	36.0	35.0	41.9	41.2	20.8	28.0	30.0	27.8
Qwen3-32B	5.3	27.0	22.5	4.5	14.8	56.0	43.0	52.5	40.0	47.9	59.5	27.2	48.0	44.9	35.9
Tool-Use Post-Trained Models (7B–8B)															
TOUCAN-7B	0.5	6.9	3.0	0.8	2.8	34.3	27.8	13.4	21.1	24.1	30.0	10.1	27.0	22.4	16.4
Simulator-8B	0.0	1.2	4.8	1.0	1.8	6.4	6.0	2.0	7.0	5.3	49.3	15.1	**51.0**	38.5	15.2
ScaleEnv-8B	3.0	**26.3**	23.8	7.0	15.0	–	–	–	–	–	50.9	27.2	37.5	38.5	–
AWM-8B	1.5	16.0	19.2	4.2	10.2	50.8	42.1	41.3	36.8	42.7	41.3	18.2	34.0	31.2	28.0
EnvFactory-8B	2.2	24.0	22.0	10.0	14.6	57.4	46.5	43.0	29.4	44.1	43.9	13.2	44.0	33.7	30.8
EnvScaler-8B	**6.0**	20.8	29.5	7.0	15.8	66.0	46.5	**48.8**	39.4	50.2	48.0	16.2	34.5	32.9	33.0
EnvACE-8B	**6.0**	24.0	27.0	7.0	16.0	53.1	43.8	45.1	40.9	45.7	48.9	17.3	44.0	36.7	32.8
SPADE-8B	0.0	5.0	1.0	0.0	1.5	58.0	43.2	36.0	30.0	41.8	37.8	21.2	29.5	29.5	24.3
BehaviorPilot-8B	5.0	26.0	**35.0**	**16.0**	**20.5**	**70.0**	**56.5**	42.5	**44.0**	**53.3**	**60.5**	**43.9**	40.0	**48.1**	**40.6**

Table 2: Performance across behavior-guided refinement rounds on BFCL and τ^2-Bench. Initial uses the initial synthesized data. Task reports approximate cumulative task numbers (k = 1,000). Bold values indicate column-wise best results.

Setting	Tasks	BFCL					τ^2-Bench			
		Base	Long Ctx	Miss Func	Miss Param	Avg.	Retail	Telecom	Airline	Avg.
Initial	˜4k	**68.5**	44.0	37.5	44.0	48.5	59.1	33.3	42.0	44.8
+1 round	˜6k	60.0	53.0	38.0	**45.0**	49.0	60.2	27.0	**51.9**	46.4
+2 rounds	˜8k	62.0	52.5	41.0	44.0	49.9	64.2	**44.0**	34.0	47.4
+3 rounds	˜10k	65.0	**58.0**	**46.0**	40.0	**52.3**	**65.8**	42.1	44.2	**50.7**

benchmarks indicates that behavior-guided training improves tool-use capabilities across diverse tasks and interaction settings.

Compared with Qwen3 models, BehaviorPilot delivers substantial gains at the same model scale. It improves over Qwen3-8B by 12.8 points overall (40.6 versus 27.8). More importantly, BehaviorPilot also surpasses the substantially larger Qwen3-32B model by 4.7 points overall. The advantage holds consistently across all three benchmarks, with improvements of 5.7 points on VitaBench, 5.4 points on BFCL V4 Multi-Turn, and 3.2 points on τ^2-Bench. These results demonstrate that behavior-guided post-training can enable an 8B model to outperform a much larger off-the-shelf model on diverse tool-use tasks.

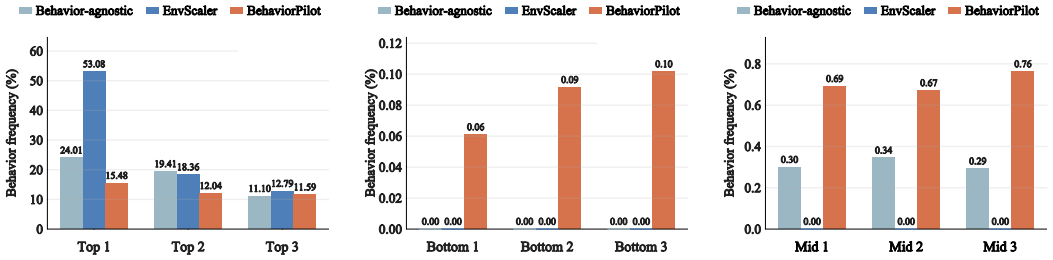
We further compare BehaviorPilot with tool-use post-trained baselines. EnvScaler-8B and EnvACE-8B are the two strongest prior methods in overall score, achieving 33.0 and 32.8, respectively. BehaviorPilot outperforms EnvScaler-8B by 7.6 points overall, including gains of 4.7 points on VitaBench, 3.1 points on BFCL V4 Multi-Turn, and 15.2 points on τ^2-Bench. It also exceeds EnvACE-8B by 7.8 points overall, with improvements of 4.5, 7.6, and 11.4 points on the three benchmarks, respectively. These results show that BehaviorPilot provides consistent improvements over strong tool-use post-training approaches across different evaluation settings.

4.4 Model Performance during Iterative Behavior-Guided Refinement

We perform three rounds of behavior-guided refinement, using behavior-level validation to guide the synthesis of approximately 2,000 additional tasks per round and updating the model through SFT. This expands the training data from approximately 4,000 to 10,000 tasks. Table 2 shows that both benchmark averages improve after every round: BFCL increases from 48.5 to 52.3, and τ^2-Bench from 44.8 to 50.7. BFCL gains are concentrated in long-context and missing-function tasks, whose scores rise from 44.0 to 58.0 and from 37.5 to 46.0, respectively. On τ^2-Bench, all three domains finish above their initial scores, with the largest absolute gain in telecom. The final round achieves the highest average on both benchmarks. Together, these results support iterative behavior-

Table 3: Performance across synthesis strategies, environment counts, and task budgets on BFCL and τ^2-Bench. Behavior-agnostic denotes synthesis without explicit behavioral guidance. Bold values indicate column-wise best results within each block.

Setting	Env.	Task	BFCL					τ^2-Bench			
			Base	Long Ctx	Miss Func	Miss Param	Avg.	Retail	Telecom	Airline	Avg.
Qwen3-8B	–	–	55.5	41.0	36.0	35.0	41.9	41.2	20.8	28.0	30.0
Joint environment and task scaling											
Behavior-agnostic	100	4k	62.7	43.1	36.2	27.4	42.4	27.2	**16.7**	24.0	22.6
Behavior-agnostic	200	8k	66.9	**46.5**	40.0	27.0	45.1	32.5	14.9	20.0	22.5
Behavior-agnostic	300	12k	67.3	46.1	39.2	**28.4**	45.3	30.7	13.2	**26.0**	23.3
Behavior-agnostic	400	16k	**67.4**	**46.5**	**40.2**	**28.4**	**45.6**	**36.8**	14.0	22.0	**24.3**
Behavior guidance and environment diversity											
Behavior-agnostic	400	4k	66.5	41.1	37.3	28.5	43.4	31.8	15.5	14.3	20.5
BehaviorPilot	100	4k	**68.5**	**44.0**	**37.5**	**44.0**	**48.5**	**59.1**	**33.3**	**42.0**	**44.8**



(a) Most frequent behaviors. (b) Least frequent behaviors. (c) Near-median behaviors.

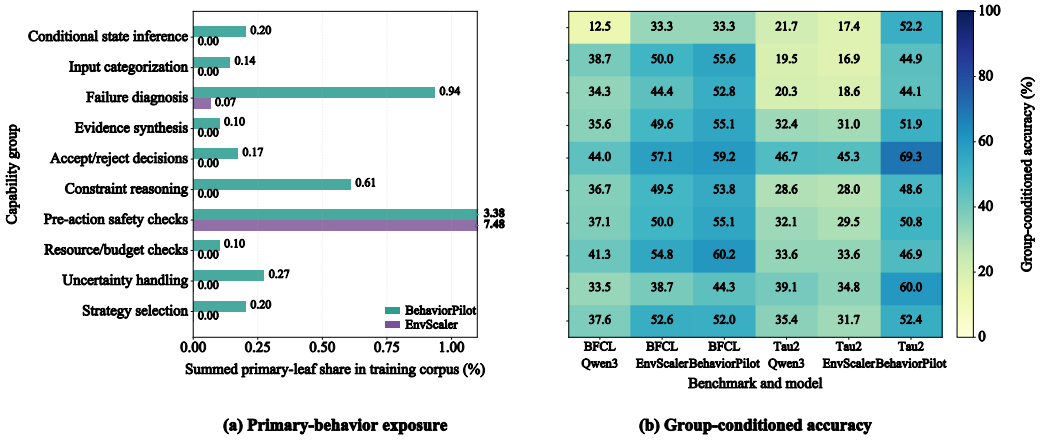
Figure 5: The distribution of behavior frequencies across synthesis methods. Frequencies are percentages of trajectories assigned to each of the 40 level-2 subcategories as their primary behavior. Each corpus independently selects its top-3, bottom-3, and three closest-to-median subcategories; aligned bars may represent different subcategories. Behavior-agnostic uses 400 environments.

guided supplementation as a practical way to extend the benefits of initial data synthesis and sustain improvements in aggregate tool-use performance.

4.5 Effect of Behavior Guidance and Environment Diversity

We analyze whether increasing environment diversity and task numbers can substitute for explicit behavior guidance. We first jointly scale behavior-agnostic synthesis from 100 environments and 4,000 tasks to 400 environments and 16,000 tasks. Table 3 shows diminishing BFCL gains: the first doubling raises the average from 42.4 to 45.1, whereas the second yields only 45.6. The τ^2-Bench average improves only modestly overall, from 22.6 to 24.3, indicating limited benefits from increasing synthesis scale alone. To assess environment diversity separately, we fix the trajectory budget at 4,000 and increase the environment count from 100 to 400. This improves BFCL only from 42.4 to 43.4 and reduces τ^2-Bench from 22.6 to 20.5. In contrast, BehaviorPilot achieves 48.5 and 44.8 with 100 environments and 4,000 tasks, outperforming even the largest behavior-agnostic setting. Within the evaluated range, neither joint scaling nor greater environment diversity recovers the gains from explicit behavior guidance.

To better understand these performance differences, we profile behavior frequencies in data produced by BehaviorPilot, behavior-agnostic synthesis, and EnvScaler, considering the three most frequent behaviors, the three least frequent behaviors, and the three closest to the median frequency. Figure 5 shows that BehaviorPilot exhibits lower head concentration, nonzero frequencies for the selected least frequent behaviors, and higher frequencies near the median. These patterns indicate that behavior guidance extends coverage beyond frequently occurring behaviors, while a larger envi-



(a) Primary-behavior exposure **(b) Group-conditioned accuracy**

Figure 6: Training behavior exposure and cross-benchmark performance for ten selected capability groups. (a) Primary-behavior exposure in the training corpora. (b) Group-conditioned accuracy on BFCL v4 and τ^2-Bench.

ronment pool can still leave behavioral coverage uneven. Together, these results highlight the value of explicitly controlling behavioral coverage rather than relying on synthesis scale alone.

4.6 Behavior-Level Analysis of Cross-Benchmark Performance

We analyze behavior-level performance to relate capability gains to training coverage. We consolidate our 100 leaf-level behaviors into semantically related capability groups and compare Qwen3-8B, EnvScaler-8B, and BehaviorPilot-8B on all tasks involving any member behavior, separately on BFCL v4 and τ^2-Bench. Training behaviors are re-identified from actual trajectories, assigning one primary behavior per trajectory; group exposure is the fraction assigned to that group. We compare BehaviorPilot-8B against Qwen3-8B using two-sided exact McNemar tests (McNemar, 1947), with Benjamini–Hochberg correction over all eligible hypotheses within each benchmark at a nominal 5% false discovery rate (Benjamini & Hochberg, 1995).

BehaviorPilot covers all 100 leaf-level behaviors under primary assignment, versus 26 for EnvScaler. Figure 6 shows ten selected groups with at least 20 tasks per benchmark and significant BehaviorPilot-8B gains over Qwen3-8B on both benchmarks after correction. In these groups, EnvScaler-8B improves over Qwen3-8B on BFCL v4 but not on τ^2-Bench. Higher exposure does not guarantee higher accuracy: for pre-action safety checks, BehaviorPilot has lower exposure than EnvScaler (3.38 versus 7.48) yet higher τ^2-Bench accuracy (50.8 versus 29.5). These findings suggest that understanding cross-benchmark gains requires considering the breadth of training-behavior coverage, rather than primary exposure to individual capability groups alone.

5 Conclusion

To address the limited control over behavioral coverage in scenario-centric synthesis, we proposed **BehaviorPilot**, a behavior-guided data synthesis framework for tool-using agents. By combining behavior taxonomy induction, behavior-conditioned data synthesis, and iterative behavior-guided refinement, BehaviorPilot makes behavioral targets explicit and directs subsequent synthesis toward model weaknesses. Applied to Qwen3-8B, BehaviorPilot achieves the best benchmark-average scores among the evaluated baselines on VitaBench, BFCL V4 multi-turn, and τ^2-Bench. Notably, the resulting 8B model also surpasses Qwen3-32B in overall performance. Iterative refinement further improves tool-use performance by targeting model weaknesses in subsequent synthesis rounds. Additional analyses show that explicit behavior guidance yields larger gains than increasing environment diversity or data volume alone.

AI USE STATEMENT

In this work, we used generative AI tools to generate synthetic datasets and assist with translation between Chinese and English. We have not used generative AI tools for other tasks requiring disclosure under the ICLR 2027 AI policy. Additionally, we used generative AI tools to edit scripts, improve the grammar and readability of the manuscript, and identify relevant literature. To check the AI-assisted outputs, we cleaned the generated synthetic data, performed execution tests on the AI-assisted scripts, and verified the suggested references against the original publications. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

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. URL https://arxiv.org/abs/2506.07982.

Yoav Benjamini and Yosef Hochberg. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal statistical society: series B (Methodological)*, 1995.

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

Guanting Dong, Junting Lu, Junjie Huang, Wanjun Zhong, Longxiang Liu, Shijue Huang, Zhenyu Li, Yang Zhao, Xiaoshuai Song, Xiaoxi Li, et al. Agent-world: 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, et al. Towards general agentic intelligence via environment scaling. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 17610–17621, 2026.

Wei He, Yueqing Sun, Hongyan Hao, Xueyuan Hao, Zhikang Xia, Qi Gu, Hui Su, and Xunliang Cai. VitaBench: Benchmarking LLM agents with versatile interactive tasks in real-world applications. In *International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=rtcX9qOBaz.

Mengkang Hu, Pu Zhao, Can Xu, Qingfeng Sun, Jian-Guang Lou, Qingwei Lin, Ping Luo, and Saravan Rajmohan. Agentgen: Enhancing planning abilities for large language model based agent via environment and task generation. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 1*, pp. 496–507, 2025.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. API-Bank: A Comprehensive Benchmark for Tool-Augmented LLMs. *arXiv preprint arXiv:2304.08244*, 2023. URL https://arxiv.org/abs/2304.08244.

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

Bo Liu, Simon Yu, Yiding Jiang, Ao Qu, Andrew Zhao, Zichen Liu, Junsu Kim, Zijian Zhou, Seungone Kim, Tongzheng Ren, et al. 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, Wang Xinzhi, 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. In *International Conference on Learning Representations*, 2025.

Zuxin Liu, Thai Hoang, Jianguo Zhang, Ming Zhu, Tian Lan, Shirley Kokane, Juntao Tan, Weiran Yao, Zhiwei Liu, Yihao Feng, Rithesh Murthy, Liangwei Yang, Silvio Savarese, Juan Carlos Niebles, Huan Wang, Shelby Heinecke, and Caiming Xiong. APIGen: Automated Pipeline for Generating Verifiable and Diverse Function-Calling Datasets. *arXiv preprint arXiv:2406.18518*, 2024. URL https://arxiv.org/abs/2406.18518.

Jiarui Lu, Thomas Holleis, Yizhe Zhang, Bernhard Aumayer, Feng Nan, Haoping Bai, Shuang Ma, Shen Ma, Mengyu Li, Guoli Yin, et al. Toolsandbox: A stateful, conversational, interactive evaluation benchmark for llm tool use capabilities. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 1160–1183, 2025.

Quinn McNemar. Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika*, 1947.

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

Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. The berkeley function calling leaderboard (BFCL): From tool use to agentic evaluation of large language models. In *Proceedings of the 42nd International Conference on Machine Learning*, Proceedings of Machine Learning Research. PMLR, 2025. URL https://proceedings.mlr.press/v267/patil25a.html.

Akshara Prabhakar, Zuxin Liu, Ming Zhu, Jianguo Zhang, Tulika Manoj Awalgaonkar, Shiyu Wang, Zhiwei Liu, Haolin Chen, Thai Hoang, Juan Carlos Niebles, et al. Apigen-mt: Agentic pipeline for multi-turn data generation via simulated agent-human interplay. *Advances in Neural Information Processing Systems*, 2025.

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

Qwen Team. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025. URL https://arxiv.org/abs/2505.09388.

Qwen Team. Qwen3.6-27B: Flagship-level coding in a 27B dense model, April 2026. URL https://qwen.ai/blog?id=qwen3.6-27b.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language Models Can Teach Themselves to Use Tools. *arXiv preprint arXiv:2302.04761*, 2023. URL https://arxiv.org/abs/2302.04761.

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

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. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 8326–8357, 2026.

Dunwei Tu, Hongyan Hao, Hansi Yang, Yihao Chen, Yi-Kai Zhang, Zhikang Xia, Yu Yang, Yueqing Sun, Xingchen Liu, Furao Shen, et al. Scaleenv: Scaling environment synthesis from scratch for generalist interactive tool-use agent training. *arXiv preprint arXiv:2602.06820*, 2026.

Weixun Wang, Shaopan Xiong, Gengru Chen, Wei Gao, Sheng Guo, Yancheng He, Ju Huang, Jiaheng Liu, Zhendong Li, Xiaoyang Li, et al. Reinforcement learning optimization for large-scale learning: An efficient and user-friendly scaling library. *arXiv preprint arXiv:2506.06122*, 2025.

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.

Zhiheng Xi, Jixuan Huang, Chenyang Liao, Baodai Huang, Honglin Guo, Jiaqi Liu, Rui Zheng, Junjie Ye, Jiazheng Zhang, Wenxiang Chen, Wei He, Yiwen Ding, Guanyu Li, Zehui Chen, Zhengyin Du, Xuesong Yao, Yufei Xu, Jiecao Chen, Tao Gui, Zuxuan Wu, Qi Zhang, Xuanjing Huang, and Yu-Gang Jiang. AgentGym-RL: Training LLM Agents for Long-Horizon Decision Making through Multi-Turn Reinforcement Learning. *arXiv preprint arXiv:2509.08755*, 2025. URL https://arxiv.org/abs/2509.08755.

Minrui Xu, Zilin Wang, Mengyi Deng, Zhiwei Li, Zhicheng Yang, Xiao Zhu, Yinhong Liu, Boyu Zhu, Baiyu Huang, Chao Chen, et al. Envfactory: Scaling tool-use agents via executable environments synthesis and robust rl. *arXiv preprint arXiv:2605.18703*, 2026a.

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

Zishan Xu, Zhiyuan Yao, Yuxin Chen, Yifu Guo, Zhengxi Lu, Yuquan Lu, Jinyang Huang, Yan Xu, Yasheng Wang, Weinan Zhang, et al. Envace: Internalizing environment dynamics via world rehearsal for agentic reinforcement learning. *arXiv preprint arXiv:2608.06197*, 2026b.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models. *arXiv preprint arXiv:2210.03629*, 2022. URL https://arxiv.org/abs/2210.03629.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. In *International Conference on Learning Representations*, 2025.

Zhaochen Yu, Ling Yang, Jiaru Zou, et al. Demystifying reinforcement learning in agentic reasoning. *arXiv preprint arXiv:2510.11701*, 2025.

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

Shaokun Zhang, Yi Dong, Jieyu Zhang, Jan Kautz, Bryan Catanzaro, Andrew Tao, Qingyun Wu, Zhiding Yu, and Guilin Liu. Nemotron-research-tool-n1: Exploring tool-using language models with reinforced reasoning. In *International Conference on Learning Representations*, 2026.

Yuze Zhao, Jintao Huang, Jinghan Hu, Xingjun Wang, Yunlin Mao, Daoze Zhang, Zeyinzi Jiang, Zhikai Wu, Baole Ai, Ang Wang, Wenmeng Zhou, and Yingda Chen. Swift:a scalable lightweight infrastructure for fine-tuning, 2024. URL https://arxiv.org/abs/2408.05517.

APPENDIX

A PROMPT TEMPLATES

BehaviorPilot uses a layered prompt stack rather than one monolithic prompt. During data construction, a stage-specific launcher is composed with persistent agent and skill instructions. During evaluation, the tool-using assistant and, for interactive tasks, the simulated user receive separate system prompts. Dynamic values are shown in angle brackets, except for the original batch-size placeholder {`tasks_per_round`}, which is retained verbatim. Repository locations and orchestration-framework names have been normalized for anonymous review without changing the behavioral instructions.

Stage	Model role	Dynamically supplied context
Environment synthesis	Generator	Scenario, scope, scale, and language
Behavior planning	Generator	Environment artifacts and behavior catalog
Task synthesis	Generator	Behavior plan, existing tasks, coverage report
Task QA	Independent verifier	Generated files on disk only
Interactive-task synthesis	Generator/verifier	Scenario plan and optional user tools
Single-turn rollout	Assistant	Task request, output shape, policy, tool schemas
Interactive rollout	Assistant and user	User scenario, policies, optional user tools

Table 4: Composition of the BehaviorPilot prompt stack.

A.1 ENVIRONMENT SYNTHESIS

The environment generator is instructed to create the four environment artifacts in a fixed order and to close the loop with executable validation. The following is a compact English rendering of the persistent instruction; the full source additionally specifies local import and file-layout checks.

Environment synthesis: persistent instruction

```
You are an "Env Generator". Your goal is to create a new environment
(env)
in the repository that can be loaded by EnvRegistry and used by
evaluation
scripts. Aim to produce a runnable environment in a single end-to-end
pass.

Generate the artifacts strictly in the following order:
1. Use prd-gen to generate PRD.md, specifying entities, fields,
business
   processes, read/write boundaries, and determinism constraints.
2. Use data-model-gen to generate data_model.py. Fields must support
stable
   JSON serialization, and only one environment.db.DB subclass is
   allowed.
3. Use database-gen to generate data.json. The data must pass
   DB.model_validate, IDs must be unique, foreign-key references must
   resolve, and no real sensitive data may be included.
4. Use toolkit-gen to generate tools.py. Tool parameters must be
compatible
   with the OpenAI tool schema. A side-effect-free submit_result must
   be
   provided. WRITE tools must not use time or random numbers to
   generate
   persistent values.

After generation, run import checks, data validation, and tool schema
```

Environment synthesis: persistent instruction (continued)

```
self-checks. If any step fails, return only to the corresponding stage
and
apply minimal fixes until the environment can be loaded and evaluated.
Do
not generate <STANDARD_TASK_DIR>/* during environment synthesis.
```

A.2 BEHAVIOR-CONDITIONED TASK SYNTHESIS

The planning prompt defines behavior as the primary coverage axis.

Behavior planning: launcher

```
Use the behavior-coverage-task-generator agent, but perform planning
only in
this round. Do not generate any tasks.
The target directory is fixed at <ENV_DIR>. Do not read or modify
<SOURCE_ENV_COLLECTION>.
Use the behavior-coverage-gen skill: read PRD.md, data_model.py,
data.json,
and tools.py; determine the supported/unsupported behavior leaves; and
write
them to behavior_plan.json. In slot_plan, use primary_leaf_ids to plan
slots. Write only behavior_plan.json to disk; do not create tasks.
```

For each batch, the generator then receives the following launcher. The persistent agent instruction additionally enforces natural task wording, tool-only reference solutions, answer-and-state verification, and truthful behavior sidecars.

Task synthesis: batch launcher

```
Use the behavior-coverage-task-generator agent. In this round,
generate
{tasks_per_round} tasks only under <ENV_DIR>. Do not read or modify
<SOURCE_ENV_COLLECTION>.
First read behavior_plan.json, the existing <STANDARD_TASK_DIR>/
directory,
and behavior_coverage_report.md (if it exists). Select the
{tasks_per_round}
highest-priority slots for which tasks have not yet been generated.
Each task generated in this round must include task.json, solution.py,
verify.py, and behaviors.json.
After generation, run python verify.py, <TASK_SMOKE_TEST_COMMAND>, and
<TASK_AUDIT_COMMAND> for each task. Require errors=0. Do not generate
more
than {tasks_per_round} tasks.
```

Independent quality assurance is run in a fresh model process using only the artifacts on disk:

Task QA: independent verifier

```
Use the task-verify skill. The target directory is fixed at <ENV_DIR>.
Do
not read or modify <SOURCE_ENV_COLLECTION>.
Check only recently added or not-yet-verified tasks. If recently added
tasks
cannot be identified, check all tasks.
Only repairs to <TASK_DIR>/task.json, solution.py, and verify.py are
allowed. Do not modify PRD.md, data_model.py, data.json, or tools.py.
Check the following:
```

Task QA: independent verifier (continued)

```
- Alignment between task_query and submit_result_example in task.json.
- The recursive observable contract between submit_result_example and
  verify.py.
- verify_with_db_and_submission.
- Whether the solution interacts with the environment only through
tools.
- Whether the primary behavior in behaviors.json is aligned with
task.json.
For each task, run python verify.py, <TASK_SMOKE_TEST_COMMAND>, and
<TASK_AUDIT_COMMAND>.
Classify the results as VERIFY_CONTRACT, VERIFY_TOO_STRICT,
TASK_UNREACHABLE, SOLUTION_DRIFT, MODEL_BEHAVIOR, or
UPSTREAM_REQUIRED.
```

A.3 SINGLE-TURN ROLLOUT

The tool-using assistant receives the following system message:

Single-turn rollout: assistant system prompt

```
You are a helpful assistant designed to accomplish user tasks using a
set of
tools. You must use the provided tools to interact with the
environment.
Before calling any WRITE tools, you must ensure the user has provided
enough
information and that the target is uniquely identified. If information
is
missing, the target is ambiguous, required safety checks are not
satisfied,
or the system lacks required capability, do NOT call any tools
(including
`submit_result`). Instead, reply with plain text starting with one of
these
prefixes: CLARIFICATION_NEEDED: / INSUFFICIENT_TOOLS: /
UNSAFE_TO_EXECUTE: , then briefly explain why and (if applicable) ask
a single
clarification question. Only when it is safe and sufficient to proceed
should
you use tools, and when you have the final answer you may use the
`submit_result` tool to submit it and follow the result format
specified in
the user query.
```

An optional environment policy is appended to this message. Tool definitions are passed through the API's structured tool parameter. The user message is constructed as follows, omitting the final two lines when no output-shape example is supplied:

Single-turn rollout: user message template

```
<TASK_QUERY>
Submit Result Format:
<SUBMIT_RESULT_EXAMPLE>
```

A.4 INTERACTIVE ROLLOUT

Interactive evaluation uses isolated assistant and user-simulator contexts. The assistant receives the single-turn system message plus an interaction addendum that permits progressive disclosure, requires schema-valid tool calls, discourages repetitive batching, makes `submit_result` optional

unless a machine-readable answer is requested, and requires private user-side facts to be elicited from the user rather than invented.

The simulated user's system prompt is:

Interactive rollout: user-simulator system prompt

```
You are simulating a realistic user interacting with an assistant.

Core rules:
- Generate exactly one natural-language user message at a time.
- Stay in character and follow the scenario exactly.
- Do not invent tool results, database state, policy, or system
events.
- Disclose information progressively and naturally. Do NOT dump the
full
  hidden task in one message.
- In the first user message, state only the broad intent and omit at
least one
  concrete detail such as date, identifier, target name, quantity,
  approval
  detail, or output preference.
- When the assistant asks a broad/open-ended question, reveal only one
or two
  directly relevant details, then wait. When the assistant asks for a
  specific
  missing detail that is in the scenario, provide that detail.
- If the assistant asks for information that is unknown in the
scenario, say
  that you do not know and ask the assistant to check if possible.
- Continue until the task is actually satisfied.
- Do not invent private user-side results. Call the relevant private
user-side
  tool only after the assistant requests the corresponding action,
  then report
  the result naturally without exposing tool-call mechanics.
- Use the scenario's completion token only after its success condition
is met;
  otherwise use [GOAL_SATISFIED].

<scenario>
Persona: <PERSONA>
Instructions:
- Domain: <DOMAIN>
- Reason For Call: <REASON_FOR_CALL>
- Known Info: <KNOWN_INFO_OR_NONE>
- Unknown Info: <UNKNOWN_INFO_OR_NONE>
- Task Instructions: <TASK_INSTRUCTIONS>
</scenario>
```

The user simulator may additionally receive an environment-specific user policy, private user-tool constraints, and a hidden answer example for semantic goal-completion judgment. The hidden example is never shown to the assistant; if it is valid JSON, only a value-masked output shape may be disclosed through conversation. Conversely, the assistant receives only backend tools and, where applicable, a description of private functions that it must ask the user to invoke.

Reproducibility note. Task-specific requests, environment policies, and generated tool schemas vary across instances and are released with their corresponding environment artifacts. An older MCP-based evaluation runner uses a different prompt with mandatory submission semantics and is not part of the active evaluation protocol reported here.

B Environment and Task Examples

B.1 Example E1: Airline Booking

B.1.1 Environment data model

The environment represents sellable fare inventory and the resulting booking as typed database records:

E1: Environment data model (Python)

```
from typing import List, Literal, Optional

from pydantic import BaseModel, Field, model_validator

BookingStatus = Literal["BOOKED", "CANCELLED", "COMPLETED",
"DISRUPTED"]

class FareInventory(BaseModel):
    id: str = Field(description="Stable fare-inventory identifier.")
    flight_instance_id: str
    cabin_brand_id: str
    seat_capacity: int = Field(ge=1)
    seats_available: int = Field(ge=0)
    cash_price: float = Field(ge=0)
    miles_price: int = Field(ge=0)
    currency: str

    @model_validator(mode="after")
    def _validate_capacity(self) -> "FareInventory":
        if self.seats_available > self.seat_capacity:
            raise ValueError("seats_available cannot exceed
            seat_capacity")
        return self

class Booking(BaseModel):
    id: str
    confirmation_code: str
    customer_id: str
    traveler_ids: List[str]
    segment_ids: List[str]
    status: BookingStatus
    booking_channel: str
    payment_method_reference: str
    currency: str
    total_cash_paid: float = Field(ge=0)
    total_miles_paid: int = Field(ge=0)
    booked_at: str
    last_updated_at: str
    cancellation_reason: Optional[str] = None
    refund_amount: float = Field(ge=0)
    certificate_ids: List[str] = Field(default_factory=list)
    support_case_ids: List[str] = Field(default_factory=list)
```

B.1.2 Environment tool implementation

The following is the core booking tool. It quotes the explicit traveler/fare combination, derives stable identifiers, consumes inventory, creates one segment per traveler and fare, and finally writes the booking aggregate.

E1: Booking tool implementation (Python)

```
@is_tool(ToolType.WRITE)
def create_booking(
    self,
    customer_id: str,
    traveler_ids: List[str],
    fare_inventory_ids: List[str],
    payment_method_reference: str,
) -> Booking:
    quote = self._quote(
        customer_id=customer_id,
        traveler_ids=traveler_ids,
        fare_inventory_ids=fare_inventory_ids,
    )
    payload = {
        "customer_id": customer_id,
        "traveler_ids": traveler_ids,
        "fare_inventory_ids": fare_inventory_ids,
        "payment_method_reference": payment_method_reference,
    }
    booking_id = f"booking_{self._sha16(self._stable_json(payload))}"
    if booking_id in self.db.bookings:
        raise ValueError("booking with the same deterministic payload
        already exists")

    confirmation_code = f"AIR{self._sha16('confirmation:' +
    booking_id)[:6].upper()}"
    self._consume_inventory(fare_inventory_ids, len(traveler_ids))

    segment_ids: List[str] = []
    for traveler_id in traveler_ids:
        for order, fare_inventory_id in enumerate(fare_inventory_ids,
        start=1):
            fare = self._get_fare_inventory(fare_inventory_id)
            segment_payload = {
                "booking_id": booking_id,
                "traveler_id": traveler_id,
                "fare_inventory_id": fare_inventory_id,
                "segment_order": order,
            }
            segment_id =
            f"seg_{self._sha16(self._stable_json(segment_payload))}"
            segment = ReservationSegment(
                id=segment_id,
                booking_id=booking_id,
                traveler_id=traveler_id,
                fare_inventory_id=fare_inventory_id,
                flight_instance_id=fare.flight_instance_id,
                cabin_brand_id=fare.cabin_brand_id,
                segment_order=order,
                status="BOOKED",
                base_cash_price=fare.cash_price,
                base_miles_price=fare.miles_price,
                ticket_number=self._sha16("ticket:" +
                segment_id).upper(),
                checked_bag_count=0,
            )
            self.db.reservation_segments[segment.id] = segment
            segment_ids.append(segment.id)

    booked_at = self._deterministic_timestamp("booked:" + booking_id)
    booking = Booking(
        id=booking_id,
        confirmation_code=confirmation_code,
```

E1: Booking tool implementation (Python) (continued)

```
        customer_id=customer_id,
        traveler_ids=traveler_ids,
        segment_ids=segment_ids,
        status="BOOKED",
        booking_channel="TOOL",
        payment_method_reference=payment_method_reference,
        currency=quote["currency"],
        total_cash_paid=quote["total_cash"],
        total_miles_paid=quote["total_miles"],
        booked_at=booked_at,
        last_updated_at=booked_at,
        cancellation_reason=None,
        refund_amount=0.0,
        certificate_ids=[],
        support_case_ids=[],
    )
    self.db.bookings[booking.id] = booking
    return booking
```

B.1.3 TASK INSTANCE

E1: Task instance (JSON)

```
{
  "task_query": "Book Traveler A at traveler.a@example.com on the
  August 4 Carrier A flight FLIGHT-101 from Chicago to San Francisco.
  Use the flexible economy option rather than the basic fare, and
  report the confirmation code, selected fare, and final total.",
  "submit_result_example":
  "{\"action\":\"create_booking\",\"booking_id\":\"<BOOKING_ID>\",\"c
  onfirmation_code\":\"<CONFIRMATION_CODE>\",\"traveler_count\":1,\"f
  are_inventory_id\":\"<FLEX_FARE_ID>\",\"total_cash_paid\":340}"
}
```

The task solver is tool-driven. The following excerpt is the actual control flow with task literals anonymized:

E1: Reference solver (Python)

```
SCENARIO = {
    "customer_email": "traveler.a@example.com",
    "traveler_names": ["Traveler A"],
    "origin_iata": "ORD",
    "destination_iata": "SFO",
    "service_date": "2026-08-04",
    "flight_number": "FLIGHT-101",
    "brand_name": "Flex",
}

def _run_create_booking(db, scenario):
    tools = bind_tools(db)
    customer_id = _customer_id_from_email(tools,
    scenario["customer_email"])
    customer = _customer_details(tools, customer_id)
    traveler_ids = _traveler_ids_by_name(
        tools, customer_id, scenario["traveler_names"]
    )
    origin_id = _airport_id_by_iata(tools, scenario["origin_iata"])
    destination_id = _airport_id_by_iata(tools,
    scenario["destination_iata"])
```

E1: Reference solver (Python) (continued)

```
    flight = _flight_summary_by_number(
        tools,
        origin_airport_id=origin_id,
        destination_airport_id=destination_id,
        service_date=scenario["service_date"],
        flight_number=scenario["flight_number"],
    )
    fare = _fare_summary_by_brand(tools, flight["id"],
    scenario["brand_name"])
    quote = _as_jsonable(
        tools.calculate_booking_quote(
            customer_id=customer_id,
            traveler_ids=traveler_ids,
            fare_inventory_ids=[fare["id"]],
        )
    )
    booking = _as_jsonable(
        tools.create_booking(
            customer_id=customer_id,
            traveler_ids=traveler_ids,
            fare_inventory_ids=[fare["id"]],
            payment_method_reference=customer["default_payment_method
            "],
        )
    )
    payload = {
        "action": "create_booking",
        "booking_id": booking["id"],
        "confirmation_code": booking["confirmation_code"],
        "traveler_count": len(traveler_ids),
        "fare_inventory_id": fare["id"],
        "total_cash_paid": booking["total_cash_paid"],
    }
    meta = {
        "booking_id": booking["id"],
        "traveler_count": len(traveler_ids),
        "fare_inventory_ids": [fare["id"]],
        "segment_ids": list(booking["segment_ids"]),
        "status": booking["status"],
        "expected_total_cash": booking["total_cash_paid"],
        "quote_total_cash": quote["total_cash"],
    }
    return _submit(tools, payload), meta
```

The corresponding verifier checks the durable write and inventory update:

E1: State and inventory verifier (Python)

```
def _verify_create_booking(before_db, after_db, meta):
    booking = after_db.bookings.get(meta["booking_id"])
    assert booking is not None, "created booking is missing"
    assert booking.status == meta["status"]
    assert round(float(booking.total_cash_paid), 2) == round(
        float(meta["expected_total_cash"]), 2
    )
    assert len(booking.segment_ids) == len(meta["segment_ids"])
    for fare_id in meta["fare_inventory_ids"]:
        before_fare = before_db.fare_inventories[fare_id]
        after_fare = after_db.fare_inventories[fare_id]
        assert (
            after_fare.seats_available
            == before_fare.seats_available - meta["traveler_count"]
```

E1: State and inventory verifier (Python) (continued)

```
        )
```

B.2 Example E2: Guarded Bank Transfer

B.2.1 Environment data model

E2: Environment data model (Python)

```
class Account(BaseModel):
    id: str
    account_number: str
    customer_id: str
    product_type: Literal[
        "CHECKING", "SAVINGS", "BUSINESS_OPERATING",
        "BUSINESS_RESERVE"
    ]
    status: Literal["ACTIVE", "PENDING_REVIEW", "FROZEN", "CLOSED"]
    currency: Literal["USD", "EUR", "GBP"]
    current_balance: float = Field(ge=0)
    available_balance: float = Field(ge=0)
    held_amount: float = Field(ge=0)
    daily_outbound_limit: float = Field(gt=0)
    per_transfer_limit: float = Field(gt=0)
    transaction_ids: List[str] = Field(default_factory=list)
    beneficiary_ids: List[str] = Field(default_factory=list)
    transfer_request_ids: List[str] = Field(default_factory=list)

class TransferRequest(BaseModel):
    id: str
    request_reference: str
    source_account_id: str
    destination_account_id: str
    beneficiary_id: str
    initiated_by_customer_id: str
    amount: float = Field(gt=0)
    currency: Literal["USD", "EUR", "GBP"]
    status: Literal[
        "PENDING", "SCHEDULED", "POSTED", "SETTLED", "CANCELLED",
        "REJECTED"
    ]
    requested_at: str
    execution_date: str
    description: str
    source_transaction_id: Optional[str] = None
    destination_transaction_id: Optional[str] = None
    reserve_ledger_entry_id: Optional[str] = None
```

B.2.2 Environment tool implementation

The public read tool exposes the same rule engine used by the write tool:

E2: Transfer-rule validation tool (Python)

```
@is_tool(ToolType.READ)
def validate_transfer_rules(
    self,
    source_account_id: str,
    beneficiary_id: str,
    amount: float,
    execution_date: str,
```

E2: Transfer-rule validation tool (Python) (continued)

```
) -> Dict[str, Any]:
    return self._transfer_validation_result(
        source_account_id,
        beneficiary_id,
        amount,
        execution_date,
    )
```

The write tool revalidates the request before changing balances and creating linked records:

E2: Transfer initiation tool (Python)

```
@is_tool(ToolType.WRITE)
def initiate_transfer(
    self,
    source_account_id: str,
    beneficiary_id: str,
    amount: float,
    requested_by_customer_id: str,
    execution_date: str,
    client_reference: str,
    description: str,
) -> Dict[str, Any]:
    validation = self._transfer_validation_result(
        source_account_id=source_account_id,
        beneficiary_id=beneficiary_id,
        amount=amount,
        execution_date=execution_date,
        requested_by_customer_id=requested_by_customer_id,
    )
    if not validation["eligible"]:
        raise ValueError("transfer is not eligible: " + ";
        ".join(validation["reasons"]))

    source = self._get_account(source_account_id)
    beneficiary = self._get_beneficiary(beneficiary_id)
    destination =
    self._get_account(beneficiary.destination_account_id)
    normalized_amount = validation["amount"]
    status = "PENDING" if execution_date == OPERATING_DATE else
    "SCHEDULED"

    id_payload = {
        "source_account_id": source_account_id,
        "beneficiary_id": beneficiary_id,
        "amount": normalized_amount,
        "requested_by_customer_id": requested_by_customer_id,
        "execution_date": execution_date,
        "client_reference": client_reference,
        "description": description,
    }
    transfer_hash = self._sha16(self._stable_json(id_payload))
    transfer_id = f"tr_{transfer_hash}"
    source_tx_id = f"txn_{transfer_hash}_src"
    destination_tx_id = f"txn_{transfer_hash}_dst"
    hold_id = f"led_{transfer_hash}_hold"

    source.available_balance = round(source.available_balance -
    normalized_amount, 2)
    source.held_amount = round(source.held_amount + normalized_amount,
    2)
```

E2: Transfer initiation tool (Python) (continued)

```
    transfer = TransferRequest(
        id=transfer_id,
        request_reference=f"TRF-NEW-{transfer_hash[:10].upper()}",
        source_account_id=source.id,
        destination_account_id=destination.id,
        beneficiary_id=beneficiary.id,
        initiated_by_customer_id=requested_by_customer_id,
        amount=normalized_amount,
        currency=source.currency,
        status=status,
        requested_at=self._deterministic_datetime(f"initiate:{transfe
        r_id}"),
        execution_date=execution_date,
        description=description,
        source_transaction_id=source_tx_id,
        destination_transaction_id=destination_tx_id,
        reserve_ledger_entry_id=hold_id,
    )
    self.db.transfer_requests[transfer.id] = transfer
    # The executable implementation also creates the two transaction
    records
    # and the HOLD ledger entry before returning their identifiers.
    return {
        "transfer_request_id": transfer.id,
        "request_reference": transfer.request_reference,
        "status": transfer.status,
        "source_transaction_id": source_tx_id,
        "destination_transaction_id": destination_tx_id,
        "reserve_ledger_entry_id": hold_id,
        "updated_available_balance": source.available_balance,
        "updated_held_amount": source.held_amount,
    }
```

B.2.3 TASK AND SOLVER CODE

The `submit_result_example` field specifies only the expected submission format, including field names, nesting structure, and value types. Its example values are illustrative and should not be interpreted as ground-truth answers or constraints on the submitted values. The agent must determine the actual values from the user request and tool execution results rather than copy them from the example.

E2: Task instance (JSON)


```
{
  "task_query": "Customer B at customer.b@example.com needs to send a
  same-day USD 900 payment from account ACCOUNT-A to the saved
  beneficiary BENEFICIARY-B. Confirm that the transfer is eligible,
  make the payment, and report the created request together with the
  updated hold on the source account.",
  "submit_result_example": "{\"customer\":\"Customer
  B\",\"source_account\":\"ACCOUNT-A\",\"beneficiary\":\"BENEFICIARY-
  B\",\"amount\":900.0,\"execution_date\":\"2026-04-15\",\"transfer_r
  equest_id\":\"<TRANSFER_ID>\",\"status\":\"PENDING\",\"updated_avai
  lable_balance\":900.0,\"updated_held_amount\":900.0}"
}
```

E2: Reference solver (Python)

```
def solve_transfer(tools, spec):
    customer_id =
    tools.find_customer_id_by_email(spec["customer_email"])
    customer = tools.get_customer_details(customer_id)
    account_id =
    tools.find_account_id_by_account_number(spec["account_number"])
    beneficiaries =
    tools.list_beneficiaries_by_source_account(account_id)
    beneficiary = next(
        row for row in beneficiaries
        if row["nickname"] == spec["beneficiary_nickname"]
    )
    tools.get_beneficiary_details(beneficiary["id"])

    validation = tools.validate_transfer_rules(
        source_account_id=account_id,
        beneficiary_id=beneficiary["id"],
        amount=spec["amount"],
        execution_date=spec["execution_date"],
    )
    if not validation["eligible"]:
        raise AssertionError(validation["reasons"])

    created = tools.initiate_transfer(
        source_account_id=account_id,
        beneficiary_id=beneficiary["id"],
        amount=spec["amount"],
        requested_by_customer_id=customer_id,
        execution_date=spec["execution_date"],
        client_reference=spec["client_reference"],
        description=spec["description"],
    )
    transfer =
    tools.get_transfer_request_details(created["transfer_request_id"])
    balances = tools.summarize_account_balances(account_id)
    return tools.submit_result({
        "customer": customer.full_name,
        "source_account": spec["account_number"],
        "beneficiary": spec["beneficiary_nickname"],
        "amount": spec["amount"],
        "execution_date": spec["execution_date"],
        "transfer_request_id": transfer.id,
        "status": transfer.status,
        "updated_available_balance": balances["available_balance"],
        "updated_held_amount": balances["held_amount"],
    })
```

C Successful Interactive Rollout

The following is an actual successful rollout selected from the interactive evaluation results. Assistant hidden reasoning is omitted. Visible message wording and call order are preserved; only names, contact details, record identifiers, and the private token are consistently anonymized.

C.1 Ticket Refund with Private User Confirmation

User

I need to refund my Ballet Under the Lights ticket. I don't remember which seat I bought, but I want to refund the cheaper one.

Assistant

Could you please provide the email address associated with your ticket purchase so I can locate your account?

User

Sure, my email address is user.d@example.com. I'd like to request a refund for one of the tickets I bought for the Ballet Under the Lights performance.

Assistant → Tool

```
find_customer_id_by_email({"email": "user.d@example.com"})
```

Tool → Assistant

```
customer_D
```

Assistant → Tool

```
list_orders_by_customer({"customer_id": "customer_D"})
```

Tool → Assistant


```
[
  {
    "id": "order_other_event",
    "occurrence_id": "occurrence_other_event",
    "status": "PAID",
    "total_amount": 88.0,
    "ticket_count": 1
  },
  {
    "id": "order_ballet_01",
    "occurrence_id": "occurrence_ballet_2025_05_02",
    "status": "PAID",
    "total_amount": 340.0,
    "ticket_count": 2
  }
]
```


Assistant → Tool

```
list_tickets_by_order({"order_id": "order_ballet_01"})
```

Tool → Assistant


```
[
  {
    "id": "ticket_ballet_A",
    "seat_id": "seat_B2",
    "holder_customer_id": "customer_D",
    "status": "ACTIVE",
    "price_paid": 190.0
  },
  {
    "id": "ticket_ballet_B",
    "seat_id": "seat_B3",
    "holder_customer_id": "customer_D",
    "status": "ACTIVE",
    "price_paid": 150.0
  }
]
```

Assistant

I found your order for "Ballet Under the Lights" which contains two tickets:

- Ticket 1: $190.00
- Ticket 2: $150.00 (cheaper one)

To proceed with refunding the cheaper ticket (`ticket_ballet_B`), I need you to verify the refund request on your end. Please run the private function `confirm_refund_request` with the ticket ID `ticket_ballet_B` and report the result back to me.

Private User Tool

```
confirm_refund_request({"ticket_id": "ticket_ballet_B"})
```

Private User Tool Result


```
{
  "fact_id": "confirm_refund_request:ticket_ballet_B",
  "tool_name": "confirm_refund_request",
  "source_collection": "tickets",
  "source_record_id": "ticket_ballet_B",
  "secret_kind": "REFUND_CONFIRM",
  "private_token": "<PRIVATE_TOKEN>",
  "selectors": {
    "id": "ticket_ballet_B",
    "holder_customer_id": "customer_D"
  },
  "payload": {
    "order_id": "order_ballet_01",
    "occurrence_id": "occurrence_ballet_2025_05_02",
    "holder_customer_id": "customer_D",
    "status": "ACTIVE",
    "is_refundable": true
  },
  "message": "Private confirmation that the holder wants a refund for
  this ticket.",
  "confirmation": "CONFIRMED"
}
```


User

Yes, my app confirms this ticket is refundable. The status shows is_refundable is true. Please process the refund.

Assistant → Tool

```
request_ticket_refund({
  "ticket_id": "ticket_ballet_B",
  "customer_id": "customer_D",
  "reason": "Customer requested refund for the cheaper ticket"
})
```

Tool → Assistant


```
{
  "id": "refund_01",
  "ticket_id": "ticket_ballet_B",
  "customer_id": "customer_D",
  "status": "COMPLETED",
  "reason": "Customer requested refund for the cheaper ticket",
  "requested_at": "2025-04-15T20:59:14",
  "resolved_at": "2025-04-15T20:59:14",
  "refund_amount": 150.0,
  "currency": "USD"
}
```

Assistant

Your refund has been successfully processed!

Refund Details:

- Ticket Refunded: Ballet Under the Lights (Cheaper Seat)
- Refund Amount: $150.00 USD
- Status: COMPLETED
- Refund Case ID: `refund_01`

The refund amount will be processed back to your original payment method. Your other ticket for this event remains active. Is there anything else I can assist you with?

User

Perfect, thanks.

GOAL_SATISFIED

`[GOAL_SATISFIED]`