STATEFORGE: SYNTHESIZING VERIFIABLE LONG-HORIZON DATA FOR TOOL-USING AGENTS VIA STATE-CHANGE SUPERVISION

Anonymous authors
Paper under double-blind review

ABSTRACT

Synthesizing long-horizon training data for tool-using agents requires reachable goals and verifiable outcomes. Successful tool calls do not guarantee task completion, while reference-call matching can reject valid alternative executions. We propose StateForge, a framework that unifies data synthesis, supervised fine-tuning (SFT), and reinforcement learning (RL) through state-change supervision. StateForge constructs executable environments with explicit state spaces and tools for querying and transforming states. Executing dependency-guided tool workflows yields reachable state-change targets, which ground task synthesis and task–target consistency checks. For each task, three trajectory synthesis modes preserve the same target: Direct provides complete goals, Progressive reveals goals in stages, and Recovery introduces recoverable environment perturbations. This shared target supports exact state-change matching to filter SFT trajectories and RL rewards that credit required changes and penalize unintended modifications. StateForge produces 544 environments and 1,697 tasks, with an average of 69 tool calls per task. With only 4B parameters, StateForge achieves state-of-the-art performance among specialized agent models on representative tool-use benchmarks, attaining 91.17% macro-average success on τ^2-Bench, 43.33% accuracy on DeepPlanning Shopping, and a 27.25% overall score on BFCL-v3 Multi-Turn.

1 INTRODUCTION

Tool-using agents complete long-horizon tasks through dependent actions that transform environment states. As these interactions grow longer, the effects of earlier actions shape the preconditions and outcomes of later ones, making local execution success an insufficient indicator of overall task completion. Synthesizing reliable training data therefore requires reachable task objectives and trajectories whose combined effects satisfy those objectives. Verification must account for both required changes and unintended effects, while accommodating different execution paths that realize the same target outcome. An explicit specification of task success can thus guide task construction, trajectory selection for SFT, and outcome feedback for RL.

Existing methods assess synthetic data and agent behavior through execution checks, reference matching, and rubric evaluation. Execution checks contribute to multi-stage data validation (Liu et al., 2024; 2025), but execution success alone does not establish that the combined effects of tool calls fulfill the task. Reference-call matching (Xu et al., 2026b) can penalize alternative sequences that produce the same correct state change. Natural-language rubrics (Gunjal et al., 2025; Huang et al., 2025) assess task fulfillment, but omitted criteria may leave unintended modifications unchecked, while judgment errors may misclassify execution outcomes. Figure 1 illustrates these challenges. Environment states provide a concrete basis for assessing tool effects, and prior work uses state-based checks for verification and training feedback (Song et al., 2026; Xu et al., 2026a). Building on state-based verification, we study how an execution-derived outcome specification can organize data synthesis and training. The central challenge is to connect three requirements: task descriptions must express reachable objectives, trajectories generated under different interaction conditions must realize those objectives, and training feedback must assess the resulting effects without prescribing a reference action sequence. Our approach uses a shared state-change target to connect these stages, grounding task generation and verification in the same executable outcome.

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.

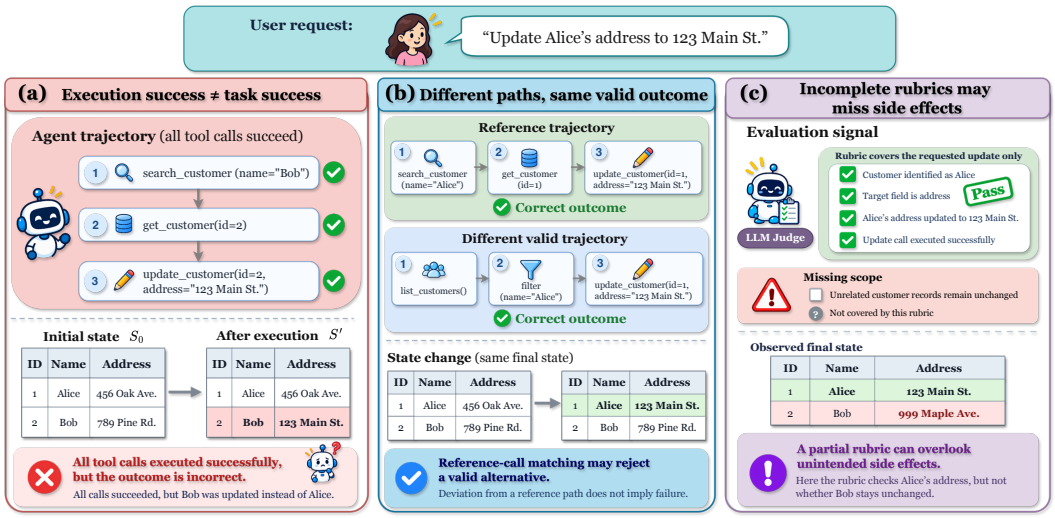


Figure 1: **Verification challenges in tool use.** For a task that requires updating Alice's address, (a) all tool calls succeed, but the agent updates Bob instead; (b) different tool-call sequences produce the same correct state change, yet reference-call matching may reject a valid alternative; and (c) a rubric covering only Alice's update can pass despite an unintended change to Bob's address.

We propose StateForge, a framework for synthesizing verifiable long-horizon tool-use data through state-change supervision. Within validated executable environments, we compose tool workflows based on dependencies among arguments, observations, and state preconditions. Executing a reference workflow from an initial state S_0 yields a reachable target state change $\Delta = \mathrm{Diff}(S_0, S_g)$, where S_g is the reference execution's terminal state. We generate a task description from the target and audit consistency: the target must cover the requested changes, and each target change must be requested or follow necessarily from the requested operations under the environment rules. The task specifies the outcome while allowing different tool-call sequences to realize the same target.

For each task, StateForge varies the interaction conditions while preserving this target. Direct trajectories provide the complete goal at the outset, Progressive trajectories reveal requirements in stages, and Recovery trajectories introduce recoverable environment perturbations. These modes provide experience in planning, incorporating additional requirements, and recovering from disruptions during long-horizon interactions. Exact state-change matching retains SFT trajectories only when all target changes are realized and no additional changes remain at termination under the environment's comparison policy. For RL under complete goal specification, the same target defines graded outcome rewards that credit required changes and penalize unintended modifications.

StateForge produces 544 environments and 1,697 tasks, with an average of 69 tool calls per task. We evaluate the resulting models on representative tool-use benchmarks covering constrained shopping planning, conversational service tasks, and multi-turn tool use. With only 4B parameters, StateForge achieves state-of-the-art performance among specialized agent models, attaining 91.17% macro-average success on τ^2-Bench, 43.33% accuracy on DeepPlanning Shopping, and a 27.25% overall score on BFCL-v3 Multi-Turn. Training-data comparisons and trajectory-mixture ablations further examine the downstream utility of the synthesized data and the contributions of the interaction modes.

This paper makes three contributions:

- We develop a framework for synthesizing executable environments with rich state spaces and diverse query and update tools. Dependency-guided workflow composition enables the construction of long-horizon tasks grounded in executable tool interactions.
- We design Direct, Progressive, and Recovery trajectory synthesis modes to provide experience in goal-directed execution, adaptation to incrementally revealed requirements, and recovery from environment perturbations, while preserving a shared task objective.

- We use execution-derived state changes to unify exact-match trajectory filtering for SFT and graded outcome rewards for RL. Experiments demonstrate improvements over the corresponding backbones at both 4B and 9B scales, achieving state-of-the-art performance among specialized agent models on representative tool-use benchmarks.

2 Related Work

2.1 Tool-Use Data Synthesis

Tool-use data synthesis spans executable environment construction, task generation, and multi-turn interaction synthesis. EnvScaler (Song et al., 2026) builds environments from task-derived themes, while Agent World Model (Wang et al., 2026a) generates code-driven, database-backed environments for agentic RL. EnvFactory (Xu et al., 2026a) constructs environments from online resources, iteratively validates tool implementations and database operations, and uses dependency graphs to compose tool workflows for query generation. APIGen (Liu et al., 2024) and ToolACE (Liu et al., 2025) generate diverse function-calling data with multiple verification steps, while APIGen-MT (Prabhakar et al., 2025) synthesizes multi-turn interactions from verified task blueprints. Trajectory2Task (Wang et al., 2026b) converts exploratory trajectories into tasks with ambiguous, changing, or infeasible user intents. StateForge combines execution-grounded task generation with variation in interaction conditions, using reachable state changes to guide task generation and task–target consistency auditing. Direct, Progressive, and Recovery synthesis vary goal disclosure and environment conditions while preserving the underlying task objective.

2.2 Training Tool-Using Agents

Training tool-using agents requires reliable demonstrations for SFT and informative rewards for RL. Verification criteria can support both demonstration filtering and outcome feedback, drawing on reference actions, natural-language criteria, or environment states. ToolRL (Qian et al., 2025) rewards agreement with reference tool names and arguments, while COVERT (Xu et al., 2026b) preserves reference calls or answers during augmentation and uses reference matching to compute rewards for standard cases. Rubrics as Rewards (Gunjal et al., 2025) and Reinforcement Learning with Rubric Anchors (Huang et al., 2025) use explicit natural-language criteria to assess task fulfillment and provide training feedback. State-based evaluation provides a complementary basis for outcome supervision: AppWorld (Trivedi et al., 2024) verifies persistent effects while allowing multiple valid solutions and detecting unexpected modifications. EnvScaler (Song et al., 2026) generates terminal-state checkers from task requirements, whereas EnvFactory (Xu et al., 2026a) combines final-state equivalence with trajectory matching and a length penalty in its RL reward. StateForge uses a common state-change comparison policy for demonstration selection and policy optimization. Exact matching filters SFT trajectories, while graded RL rewards credit required effects and penalize unintended modifications. Neither SFT filtering nor RL outcome evaluation requires agreement with a reference tool-call sequence.

3 Problem Formulation

We consider tool-use tasks and represent each task objective as a fixed target state change. A successful trajectory must realize the required changes while preserving unrelated state, although the sequence of actions may vary.

Environment and interaction. An environment E has a state space $\mathcal{S}_E$ and provides tools for reading state and modifying it within declared write scopes. At each step, the agent selects a tool and its arguments based on the available user messages and interaction history, then receives an observation. We denote the complete interaction trajectory by τ, which includes user messages, tool calls, tool responses, and the final assistant response.

Task and target definition. We represent a task as $x = (E, q^\star, S_0, \Delta)$, where $q^\star$ is the complete task description, $S_0 \in \mathcal{S}_E$ is the initial state, and Δ is the target state change. During task construction, a reference execution from S_0 reaches a terminal state $S_g \in \mathcal{S}_E$, yielding a reachable target $\Delta = \mathrm{Diff}(S_0, S_g)$, where Diff identifies additions, deletions, and modifications between two states.

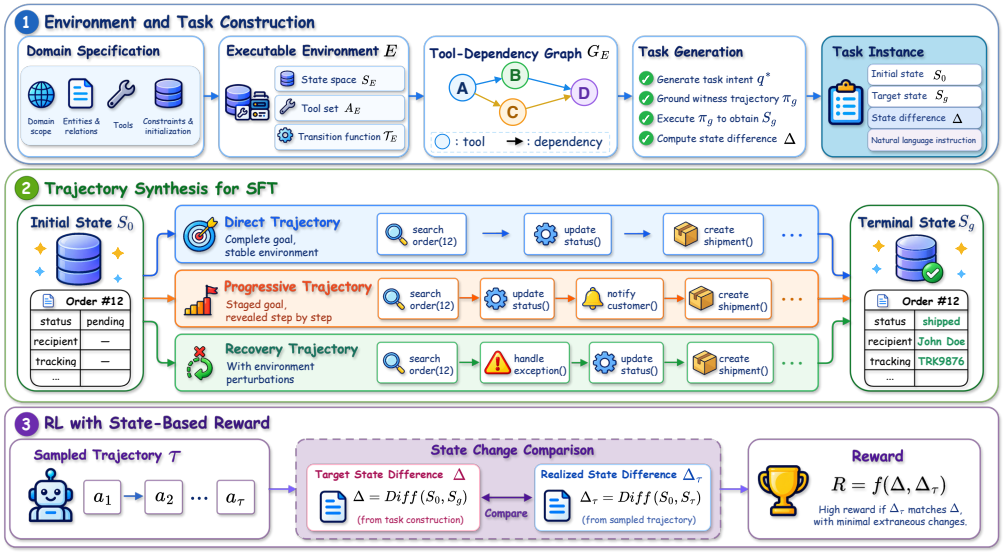


Figure 2: **Overview of StateForge.** Reference execution produces a reachable target state change $\Delta = \mathrm{Diff}(S_0, S_g)$, where S_g is the terminal state. A task–target audit verifies consistency between the task description and Δ. Direct, Progressive, and Recovery synthesis vary the interaction conditions while preserving the same target. Exact state-change matching filters SFT demonstrations, while matching provides outcome rewards for RL under the complete goal specification.

The task description expresses the objective in natural language and may be disclosed in full or in stages, while Δ remains fixed and is used for verification and reward computation without being exposed to the agent.

State-change verification. Let $S_\tau \in \mathcal{S}_E$ denote the terminal state reached under the applicable interaction conditions, including any environment interventions and subsequent agent actions. The evaluator measures the realized state change relative to the original S_0 and checks whether $\mathrm{Diff}(S_0, S_\tau) \equiv_E \Delta$. The comparison covers all state components represented in the environment schema. The relation $\equiv_E$ denotes equivalence under a fixed comparison policy: all target changes must be realized, and no additional state changes may remain at termination. The comparison rules, including recursive value comparison and permitted normalization of runtime-generated values, are detailed in Appendices B.1 and B.2. The same comparison policy is used for task construction, trajectory validation, and reward computation. Because verification assesses the resulting state changes rather than the tool-call sequence, different execution paths can satisfy the same target without reproducing the reference calls.

4 Method

As shown in Figure 2, StateForge uses an execution-derived state-change target as a shared outcome specification across task construction, trajectory synthesis, and training.

4.1 Task Construction from State Changes

We construct long-horizon tool-use tasks by composing workflows over rich executable state spaces and deriving task objectives from state changes produced through execution.

Environment synthesis. We collect environment seeds from benchmarks, skill repositories, and other task-oriented resources and construct executable environments E with explicit state spaces $\mathcal{S}_E$. For each domain, we expand the state space with additional entities, fields, relations, and permitted transitions while enforcing integrity constraints. Based on the expanded state space, we construct diverse query and update tools with explicit contracts specifying their arguments, observations, state effects, and failure conditions. Static checks and tool-sequence execution validate initialization, state

persistence, tool behavior, and complete state snapshots; environments with unresolved failures after repair are discarded. Appendix A.1 details the construction and validation procedures.

Workflow construction. The tools operate on shared entities and related state transitions, creating dependencies that we represent as a directed graph $G_E = (V_E, \mathcal{E}_E)$. Each node represents a tool, and an edge $u \to v$ indicates that u can provide an argument, identify an entity, or establish a state precondition required by v. We infer candidate dependencies from tool schemas and declared behavior, then remove edges unsupported by the tool interfaces or environment constraints. We sample workflow skeletons from G_E using bounded depth-first search to explore long dependency paths and random walks to diversify workflow structures. We retain only candidates containing at least one state-changing operation and reduce redundancy by filtering candidates with excessive overlap in tool nodes, dependency edges, or tool-chain signatures.

Reference execution and target extraction. For each candidate workflow, we instantiate an initial state $S_0 \in \mathcal{S}_E$ containing the required objects, relations, and prerequisites and validate it through the environment's initialization interface. We then finalize the workflow as a directed acyclic graph encoding the dependencies among execution steps. Following these dependencies, we instantiate and execute a reference tool-call sequence whose arguments are grounded in preceding observations, explicit workflow parameters, or S_0. A successful reference trajectory τ_r reaches terminal state S_g, yielding a reachable candidate target $\Delta = \mathrm{Diff}(S_0, S_g)$.

Task synthesis and audit. Even for long reference workflows, the net state change Δ provides an explicit outcome specification from which we generate $q^\star$, using S_0 and the execution record to ground entities, context, and constraints. The resulting task description specifies the required outcome without prescribing the reference call sequence, allowing alternative workflows. We audit task–target consistency in both directions: every change required by $q^\star$ must be represented in Δ, and every change in Δ must either be explicitly requested or necessarily follow from the requested operations under the environment rules. The audit also checks that entity references are unambiguous and that the information required for execution is accessible to the agent. Each accepted task is exported as $x = (E, q^\star, S_0, \Delta)$. The initial state, target, and comparison policy remain fixed during subsequent trajectory synthesis and training. Appendix A.2 provides detailed procedures for workflow construction, reference execution, and task auditing.

4.2 Trajectory Synthesis under a Shared State-Change Target

For each accepted task, we generate trajectories under three interaction conditions, each starting from an isolated copy of S_0 and evaluated against the same target Δ. The agent sees only disclosed task descriptions, tool schemas, and interaction history; reference terminal states, target state changes, undisclosed task descriptions, and controller plans remain hidden.

- **Direct: complete goal specification.** The agent receives the complete task description $q^\star$ at the start and interacts with the environment without intervention. Direct trajectories train the model to plan and execute tasks with fully specified goals.
- **Progressive: staged goal revelation.** The controller decomposes the complete task description into stage-specific descriptions $(q_1, \ldots, q_K)$ with cumulative state-change targets $\Delta^{(1)} \subsetneq \cdots \subsetneq \Delta^{(K)} = \Delta$. Each stage introduces additional effects from the complete target while preserving values established by earlier stages. Starting with q_1, the controller checks the current state S after each action and, for $k < K$, reveals q_{k+1} only when $\mathrm{Diff}(S_0, S) \equiv_E \Delta^{(k)}$. A valid decomposition requires the stage targets to be sequentially reachable from S_0 and to include any consequential updates required by the environment rules. Progressive trajectories train the model to incorporate new requirements while preserving completed effects.
- **Recovery: recoverable environment perturbations.** The agent receives the complete task description, while the controller perturbs the environment at designated trigger states, for example by overwriting a completed update or invalidating a prerequisite. Each intervention preserves environment invariants and keeps the original target state change Δ achievable from the perturbed state S. The controller records the intervention and exposes the resulting observation to the agent, which must adapt its execution to realize the original target. The resulting net state change is measured relative to S_0 and compared with Δ, so any deviation introduced by the intervention must

be repaired before termination. Recovery trajectories train the model to respond to environment perturbations, repair disrupted state, and replan to achieve the original objective.

Appendix C illustrates Direct, Progressive, and Recovery through concrete examples and details condition-specific validation.

Validation and SFT conversion. For each task and execution condition κ, we sample multiple candidate trajectories and record their messages, tool calls, and observations. Let $S_\tau^\kappa = \mathrm{Exec}_{E,\kappa}(S_0, \tau)$ denote the terminal state of a candidate and $\Delta_\tau^\kappa = \mathrm{Diff}(S_0, S_\tau^\kappa)$ its realized state change. We omit the superscript κ when the interaction condition is clear. A candidate passes validation if $\mathrm{Accept}_x(\tau, \kappa) = \mathbf{1}[\Delta_\tau^\kappa \equiv_E \Delta] \cdot \mathrm{Valid}_x(\tau, \kappa)$, where $\mathrm{Valid}_x(\tau, \kappa)$ captures protocol, replay, and interaction-validity checks. Among the candidates that pass validation, we select the most efficient trajectory for each task and execution condition. Each assistant turn in a selected trajectory forms an SFT example conditioned only on the preceding interaction history, with loss applied to that turn alone. Appendix C.4 details the validation, selection, and conversion procedures.

4.3 Reinforcement Learning with State-Change Rewards

We use state-change matching as the primary reward signal for RL, extending the exact-match criterion used in SFT validation to provide feedback on task outcomes. This outcome signal is computed directly from environment states, without reference-call matching or an additional reward model.

State-change reward. The state-change reward compares the state change realized by a rollout with Δ using the same comparison policy as in SFT validation. After normalization and record alignment, let $\mathcal{P}_x$ and $\mathcal{P}_\tau$ denote the paths changed in the target and rollout under the same difference rules. The set $\mathcal{P}_{\mathrm{match}}$ contains paths in $\mathcal{P}_x \cap \mathcal{P}_\tau$ whose resulting values or absence match the target changes. Paths changed only by the rollout form $\mathcal{P}_{\mathrm{unexpected}} = \mathcal{P}_\tau \setminus \mathcal{P}_x$. We assign them a penalty $U = \sum_{p \in \mathcal{P}_{\mathrm{unexpected}}} \max\{1, L_p(S_0), L_p(S_\tau)\}$, where L_p counts the logical leaves under path p, so unexpected subtree additions or deletions are weighted by their size. For nonempty targets with $|\mathcal{P}_x| > 0$, the reward is $R_{\mathrm{goal}} = |\mathcal{P}_{\mathrm{match}}|/(|\mathcal{P}_x| + U)$. Correctly realized target effects receive credit, while missing or incorrect effects do not, and unintended changes increase the penalty U. The reward lies in $[0, 1]$ and equals one if and only if the realized state change exactly matches Δ under the comparison policy. Thus, full state-change reward requires all target effects to be realized with no additional changes remaining at termination.

Policy optimization. We optimize the policy using Group Relative Policy Optimization (GRPO; Shao et al., 2024), with the state-change reward as the primary task signal. We also use a low-weight auxiliary format reward, $R_{\mathrm{format}} \in [0, 1]$, for parseable actions, structurally valid tool calls, and a valid termination format. The final training reward is $R = 0.9R_{\mathrm{goal}} + 0.1R_{\mathrm{format}}$. For each task, we sample a group of rollouts from the current policy, each starting from isolated copies of S_0, and compute relative advantages from the terminal rewards. Appendix B details the state-comparison rules, the convention for empty targets, and the auxiliary format checks.

5 Experiments

5.1 Experimental Setup

Training data and implementation. Our dataset contains 544 executable environments and 1,697 tasks. From these tasks, we construct 34,782 SFT examples from Direct, Progressive, and Recovery trajectories in a 1:1:1 ratio. We initialize StateForge-4B and StateForge-9B from Qwen3.5-4B and Qwen3.5-9B, respectively. For each model scale, we first perform SFT on the constructed examples and then apply RL to the SFT checkpoint. We denote the resulting checkpoints with the suffixes `-SFT` and `-RL`, respectively. The SFT training parameters are provided in Appendix D.2.

Benchmarks. **DeepPlanning Shopping** (Zhang et al., 2026) requires construction of shopping carts under product, promotion, and budget constraints. We report complete-cart accuracy (Acc.), accuracy at three constraint levels (L1–L3), and Match Score for partial product matching. **τ^2-Bench** (Barres et al., 2025) evaluates conversational service tasks in the Airline, Retail, and Telecom domains. We report the success rate for each domain and the unweighted mean across domains

Table 1: **Main results** (%, ↑). Bold indicates the best result among StateForge checkpoints in each column, including ties. Shopping level scores appear in Appendix D.5.

Model	τ^2-Bench				DeepPlanning		BFCL-v3 Multi-Turn				
	Airline	Retail	Telecom	Macro	Acc.	Match	Base	MF	MP	LC	Overall
Proprietary models											
GPT-5.2 High	73.79	76.49	84.21	78.16	61.40	88.40	27.50	16.50	20.50	20.00	21.13
Claude Opus 4.5	84.00	79.60	92.30	85.30	51.40	83.70	21.50	18.00	19.50	16.50	18.88
Gemini-3-Pro-Pre	80.50	75.90	91.00	82.47	54.00	83.40	42.00	40.00	38.50	32.50	38.25
General-purpose open-weight models											
DeepSeek-V3.2	63.80	81.10	96.20	80.37	54.00	84.00	35.00	31.50	24.50	25.00	29.00
Qwen3.5-4B (Base)	76.00	77.20	100.00	84.40	35.83	74.36	33.50	13.00	25.50	29.00	25.25
Qwen3.5-9B (Base)	80.00	86.80	100.00	88.93	40.83	77.47	34.50	8.00	27.50	30.00	25.00
Agent models											
ToolVerse-TARA-8B	26.00	50.00	21.10	32.37	—	—	—	—	—	—	—
AWM-14B	31.50	63.60	17.76	37.62	9.17	49.33	21.00	8.50	13.50	14.50	14.38
EnvScaler-8B	50.00	53.51	14.91	39.47	10.83	54.08	29.50	15.00	18.00	18.00	20.13
EnvFactory-8B	44.00	43.86	13.16	33.67	4.17	28.39	15.00	9.50	9.00	9.50	10.75
EnvCraft-Tool-9B	42.00	40.35	26.50	36.28	—	—	—	—	—	—	—
ScaleEnv-32B	48.00	63.60	30.90	47.50	—	—	—	—	—	—	—
Agent-World-14B	52.00	74.50	56.10	60.87	—	—	—	—	—	—	—
AgentSkiller-14B	56.00	77.20	91.20	74.80	10.00	36.39	8.00	2.00	9.50	7.00	6.63
TOUCAN-32B	22.00	52.60	20.20	31.60	8.33	48.52	34.50	20.50	29.00	23.50	26.88
StateForge models											
StateForge-4B-SFT	78.00	78.90	99.10	85.33	42.50	78.57	37.00	12.00	**27.00**	32.00	27.00
StateForge-4B-RL	**84.00**	89.50	**100.00**	91.17	43.33	77.84	**39.00**	13.50	26.00	30.50	**27.25**
StateForge-9B-SFT	**84.00**	89.47	98.25	90.57	45.83	78.21	36.00	13.50	23.50	33.50	26.63
StateForge-9B-RL	**84.00**	**91.23**	**100.00**	**91.74**	**46.67**	**80.03**	36.00	**14.00**	22.50	**34.50**	26.75

(Macro). **BFCL-v3 Multi-Turn** (Mao et al., 2024) evaluates the Base, Missing Functions (MF), Missing Parameters (MP), and Long Context (LC) categories. We report these category scores and their unweighted mean as the multi-turn Overall score (BFCL-MT).

Baselines. We compare against proprietary models GPT-5.2 High (OpenAI, 2025), Claude Opus 4.5 (Anthropic, 2025), and Gemini-3-Pro-Pre (Google DeepMind, 2026); general-purpose open-weight models DeepSeek-V3.2 (DeepSeek-AI, 2025) and the two Qwen3.5 backbones; and nine specialized agent models. The agent baselines include ToolVerse-TARA-8B (Zhou et al., 2026), AWM-14B (Wang et al., 2026a), EnvScaler-8B (Song et al., 2026), EnvFactory-8B (Xu et al., 2026a), EnvCraft-Tool-9B (Zeng et al., 2026), ScaleEnv-32B (Tu et al., 2026), Agent-World-14B (Dong et al., 2026), AgentSkiller-14B (Sun et al., 2026), and TOUCAN-32B (Xu et al., 2025).

Reporting. All scores are computed according to the success criteria defined by the corresponding benchmarks and are reported as percentages. Published results on τ^2-Bench are taken from the corresponding papers, whereas we evaluate the Qwen3.5 backbones and all models trained in this work. Results on DeepPlanning Shopping and BFCL-v3 are obtained using benchmark environments configured and executed by us. "–" denotes results that are not reported. Score differences are computed before rounding and reported in percentage points (pp). Appendix D.3 gives the metric definitions.

5.2 Main Results

StateForge achieves strong performance with only 4B parameters. StateForge-4B-RL outperforms the reported specialized agent baselines on all three aggregate metrics, achieving 91.17% τ^2 Macro, 43.33% Shopping accuracy, and 27.25% BFCL-MT. On BFCL-MT, it exceeds TOUCAN-32B's 26.88% score with substantially fewer parameters. StateForge-9B-RL further improves τ^2 Macro to 91.74%, the highest score in the comparison, and Shopping accuracy to 46.67%.

Synthesized interaction data provides gains beyond model scaling. StateForge-4B-SFT outperforms the larger Qwen3.5-9B Base model on both Shopping accuracy and BFCL-MT. On Shopping,

Table 2: **Comparison of synthesized environments and tool-chain statistics.** Task counts refer to task instances rather than SFT records. Approximate baseline lengths are marked with ≈.

Dataset	# Envs	# Tasks	Avg. tools / env	Avg. tool-chain length
EnvFactory (Xu et al., 2026a)	85	2,575	9.91	≈ 11.9
EnvScaler (Song et al., 2026)	191	11,572	18.58	≈ 15–17
StateForge	544	1,697	**50.87**	**69**

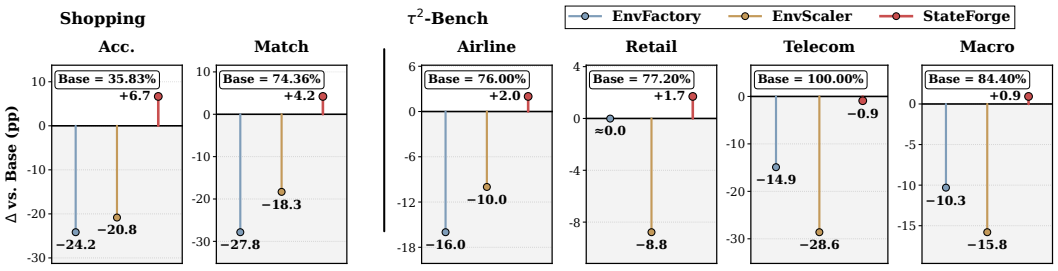


Figure 3: **Training-data comparison on Qwen3.5-4B at the SFT stage.** All three data sources use the same backbone and SFT configuration. Values show score differences from the backbone before fine-tuning (Base), measured in percentage points (pp). Positive values indicate improvements, and the zero line denotes unchanged performance. Insets report absolute Base scores. Complete absolute scores are provided in Appendix D.4, Table 5.

training the 4B model improves accuracy by 6.67 pp, exceeding the 5.00 pp gain from increasing the backbone size from 4B to 9B. On BFCL-MT, the two Base models perform similarly, while StateForge-4B-SFT improves over its backbone by 1.75 pp. These results provide evidence for the benefit of training on interaction trajectories that include task execution, progressively revealed requirements, and corrective actions.

RL with state-change supervision provides further gains. RL further improves the SFT checkpoints at both model scales, increasing τ^2 Macro by 5.83 pp for the 4B model and 1.17 pp for the 9B model. These gains suggest that feedback from the policy's own executions provides useful supervision beyond the synthesized demonstrations. By evaluating the state changes produced by the policy against the intended effects of the task, the reward links action choices to whether the required outcomes are achieved. The gains at both model scales support the use of execution-derived state changes for continued policy optimization.

5.3 Training Data Comparison

We compare the downstream utility of StateForge, EnvFactory, and EnvScaler training data using the same Qwen3.5-4B backbone at the SFT stage. When training on EnvFactory and EnvScaler data, we use the same SFT configuration as for StateForge. Base denotes the backbone before fine-tuning. Table 2 summarizes the environment and tool-chain statistics of the three data sources. Figure 3 compares downstream score changes relative to Base; complete absolute scores are reported in Appendix D.4.

StateForge data improves the utility of continued training. As shown in Figure 3, StateForge is the only training corpus that improves both Shopping accuracy and τ^2 Macro over Base under the shared backbone and SFT configuration, with gains of 6.7 pp and 0.9 pp, respectively. In contrast, EnvFactory reduces these scores by 24.2 pp and 10.3 pp, while EnvScaler reduces them by 20.8 pp and 15.8 pp. StateForge also achieves higher scores than both alternative corpora on all six reported metrics. Within τ^2-Bench, StateForge improves Airline and Retail by 2.0 pp and 1.7 pp, respectively, with a modest 0.9 pp decrease on Telecom; the corresponding Telecom decreases are 14.9 pp for EnvFactory and 28.6 pp for EnvScaler. These results show that additional interaction training does not automatically improve a strong backbone and support the downstream utility of the StateForge corpus under the evaluated SFT setup.

Table 3: **Ablation of StateForge trajectory types on Qwen3.5-4B** (%, ↑). Bold indicates the best result for each metric.

Mixture	Shopping					τ^2-Bench			
	L1	L2	L3	Acc.	Match	Airline	Retail	Telecom	Macro
Base	40.00	30.00	40.00	35.83	74.36	76.00	77.20	**100.00**	84.40
Direct	44.00	**36.00**	45.00	40.83	**79.30**	72.00	78.60	90.00	80.20
Direct + Progressive	50.00	34.00	**50.00**	**43.33**	78.75	**82.00**	76.30	96.50	84.93
Full	**54.00**	28.00	**50.00**	42.50	78.57	78.00	**78.90**	99.10	**85.33**

Broader interaction coverage and outcome verification support data quality. StateForge spans 544 executable environments, with an average of 50.87 tools per environment and 69 tool calls per chain. Progressive and Recovery trajectories extend this coverage to incrementally revealed requirements and recoverable environment perturbations, while state-change verification ties retained demonstrations to the required outcomes. The downstream gains are consistent with the utility of this combined design. However, the comparison evaluates the complete data configuration and does not isolate individual design choices or establish data efficiency under an equal training budget. Section 5.4 further examines the trajectory mixture.

5.4 Ablation Study

We ablate the StateForge trajectory mixture using the Qwen3.5-4B backbone at the SFT stage. **Direct** contains trajectories with complete task descriptions. **Direct+Progressive** adds trajectories in which task requirements are revealed across turns. **Full** additionally includes Recovery trajectories with recoverable environment perturbations. All conditions use the shared state-change target construction described in Section 4.2.

Progressive trajectories improve the utility of synthesized demonstrations. Training on Direct trajectories increases Shopping accuracy from 35.83% to 40.83%, but reduces the τ^2 Macro score from 84.40% to 80.20%. This result shows that successful execution demonstrations alone do not ensure consistent downstream gains. Adding Progressive trajectories improves the two aggregates by 2.50 pp and 4.73 pp, respectively, bringing both above Base. Progressive trajectories make the policy act before all requirements are available and adjust its execution as new requirements are revealed. The policy must relate each new instruction to the current state and preserve effects that remain required. The gains suggest that progressively revealed tasks provide useful training signals beyond complete task descriptions.

Recovery trajectories further improve the training mixture. Adding Recovery yields the highest τ^2 Macro score of 85.33%, with gains in Retail and Telecom offsetting the decrease in Airline. These trajectories introduce recoverable deviations from the expected environment state, requiring the policy to identify unmet state-change targets and take corrective actions. The gains are not uniform across benchmarks: Shopping accuracy decreases by 0.83 pp relative to Direct+Progressive. Nevertheless, the full mixture exceeds Direct-only training by 1.67 pp on Shopping and 5.13 pp on τ^2 Macro, and improves both aggregates over Base. Together, these results support combining successful execution demonstrations with progressively revealed requirements and recoverable environment perturbations in StateForge.

6 Conclusion

We presented StateForge, a framework that uses execution-derived state changes as a unified supervision signal for task construction, trajectory synthesis, and agent training. A shared state-change target supports the synthesis of Direct, Progressive, and Recovery trajectories, exact-match filtering for SFT, and graded state-change rewards for RL. Experiments on representative tool-use benchmarks show improvements over the backbones at both model scales, supporting the use of state-change supervision for training tool-using agents. Future work will extend the verification framework to accommodate multiple acceptable outcomes and constraints on intermediate behavior.

AI USE STATEMENT

Generative AI tools assisted with manuscript organization and language revision. AI-generated suggestions are not treated as scientific evidence; the authors are responsible for the technical claims, references, and final manuscript.

ETHICS STATEMENT

The framework constructs tasks in synthetic environments rather than operating on live user accounts. Generated workflows can nevertheless encode biased assumptions or omit safeguards present in deployed systems. State agreement concerns modeled effects and does not establish that an action is authorized or safe in a real application. Applying the resulting agents requires appropriate permission and privacy controls.

REPRODUCIBILITY STATEMENT

Sections 3 and 4 define the task specification and synthesis procedure, with implementation details referenced alongside the corresponding components. Appendix C provides recorded excerpts of Direct, Progressive, and Recovery trajectories, along with the trajectory validation, selection, and SFT conversion procedures. Appendix D collects experimental settings and extended results.

REFERENCES

Anthropic. Introducing Claude Opus 4.5. Official model announcement, 2025. URL https://www.anthropic.com/news/claude-opus-4-5.

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

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

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. Agent-World: Scaling real-world environment synthesis for evolving general agent intelligence, 2026. URL https://arxiv.org/abs/2604.18292.

Google DeepMind. Gemini 3 Pro model card. Official model card, 2026. URL https://deepmind.google/models/model-cards/gemini-3-pro/. Model released November 2025; model card updated May 2026.

Anisha Gunjal, Anthony Wang, Elaine Lau, Vaskar Nath, Yunzhong He, Bing Liu, and Sean Hendryx. Rubrics as rewards: Reinforcement learning beyond verifiable domains, 2025. URL https://arxiv.org/abs/2507.17746.

Zenan Huang, Yihong Zhuang, Guoshan Lu, Zeyu Qin, Haokai Xu, Tianyu Zhao, Ru Peng, Jiaqi Hu, Zhanming Shen, Xiaomeng Hu, Xijun Gu, Peiyi Tu, Jiaxin Liu, Wenyu Chen, Yuzhuo Fu, Zhiting Fan, Yanmei Gu, Yuanyuan Wang, Zhengkai Yang, Jianguo Li, and Junbo Zhao. Reinforcement learning with rubric anchors, 2025. URL https://arxiv.org/abs/2508.12790.

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. In *International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=8EB8k6DdCU.

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. In *Advances in Neural Information Processing Systems*, volume 37, 2024. doi: 10.52202/079017-1725. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/61cce86d180b1184949e58939c4f983d-Abstract-Datasets_and_Benchmarks_Track.html.

Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Jason Huang, Vishnu Suresh, Yixin Huang, Xiaowen Yu, Joseph E. Gonzalez, and Shishir G. Patil. BFCL V3: Multi-turn and multi-step function calling evaluation. UC Berkeley Gorilla project, 2024. URL https://gorilla.cs.berkeley.edu/blogs/13_bfcl_v3_multi_turn.html.

OpenAI. Introducing GPT-5.2. Official model announcement, 2025. URL https://openai.com/index/introducing-gpt-5-2/.

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, 2025. URL https://arxiv.org/abs/2504.03601.

Cheng Qian, Emre Can Acikgoz, Qi He, Hongru Wang, Xiusi Chen, Dilek Hakkani-Tur, Gokhan Tur, and Heng Ji. ToolRL: Reward is all tool learning needs. In *Advances in Neural Information Processing Systems*, volume 38, 2025. doi: 10.52202/085713-3524. URL https://papers.nips.cc/paper_files/paper/2025/hash/97c5b2707228e7e3fb67e4ecc2e0e607-Abstract-Conference.html.

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

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

Zexu Sun, Bokai Ji, Hengyi Cai, Shuaiqiang Wang, Lei Wang, Guangxia Li, and Xu Chen. AgentSkiller: Scaling generalist agent intelligence through semantically integrated cross-domain data synthesis, 2026. URL https://arxiv.org/abs/2602.09372.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. AppWorld: A controllable world of apps and people for benchmarking interactive coding agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 16022–16076, 2024. doi: 10.18653/v1/2024.acl-long.850. URL https://aclanthology.org/2024.acl-long.850/.

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

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, 2026a. URL https://arxiv.org/abs/2602.10090.

Ziyi Wang, Yuxuan Lu, Yimeng Zhang, Pei Chen, Ziwei Dong, Jing Huang, Jiri Gesi, Xianfeng Tang, Chen Luo, Qun Liu, Yisi Sang, Hanqing Lu, Manling Li, Jin Lai, and Dakuo Wang. Trajectory2Task: Training robust tool-calling agents with synthesized yet verifiable data for complex user intents. In *Proceedings of the 64th Annual Meeting of the Association for Computational*

Linguistics (Volume 1: Long Papers), pp. 44021–44044, 2026b. doi: 10.18653/v1/2026.acl-long.2037. URL https://aclanthology.org/2026.acl-long.2037/.

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, 2026a. URL https://arxiv.org/abs/2605.18703.

Siyuan Xu, Shiyang Li, Xin Liu, Tianyi Liu, Yixiao Li, Zhan Shi, Zixuan Zhang, Zilong Wang, Qingyu Yin, Jianshu Chen, Tuo Zhao, and Bing Yin. Controllable and verifiable tool-use data synthesis for agentic reinforcement learning, 2026b. URL https://arxiv.org/abs/2604.09813.

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

Yirong Zeng, Shen You, Jinhang Feng, Yufei Liu, Xiao Ding, Yutai Hou, Hao Cong, Yuxian Wang, Wu Ning, Wang Xu, and Bibo Cai. EnvCraft: Synthesizing executable environments in agentic RL for claw-like agent, 2026. URL https://arxiv.org/abs/2609.05576.

Yinger Zhang, Shutong Jiang, Renhao Li, Jianhong Tu, Yang Su, Lianghao Deng, Xudong Guo, ChenXu Lv, and Junyang Lin. DeepPlanning: Benchmarking long-horizon agentic planning with verifiable constraints. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7377–7407. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.acl-long.335. URL https://aclanthology.org/2026.acl-long.335/.

Shuaiyu Zhou, Fengpeng Yue, Zengjie Hu, Yuanzhe Shen, Chenyang Zhang, Feng Hong, Cao Liu, and Ke Zeng. ToolVerse: Unlocking massive environments and long-horizon tasks for agentic reinforcement learning, 2026. URL https://arxiv.org/abs/2607.15660.

APPENDIX OVERVIEW

The appendices follow the StateForge pipeline from environment and task construction to state-change verification, trajectory preparation, and experimental evaluation:

- **Appendix A: Environment and Task Construction Details.** Environment synthesis, tool construction, execution checks, and state coverage; workflow proposals, grounded reference execution, task-description auditing, and task export.
- **Appendix B: State-Change Verification and Reward Computation.** Changed-path extraction and matching, collection comparison and runtime-value normalization, the state-change reward and its exact-matching property, the auxiliary format reward, and policy optimization.
- **Appendix C: Trajectory Case Studies, Validation, and SFT Conversion.** Case studies of complete disclosure, successive user requests, and correction after tool errors; condition eligibility, controller events, replay and acceptance, representative selection, and information-preserving SFT conversion.
- **Appendix D: Experimental Details and Supplementary Results.** Training-data accounting, SFT configuration, evaluation sources and metrics, Shopping results by difficulty level, SFT and RL training-stage decomposition, and comparison controls and interpretation.

A ENVIRONMENT AND TASK CONSTRUCTION DETAILS

This appendix supplements Section 4.1 with implementation and validation details for environment and task construction.

A.1 ENVIRONMENT SYNTHESIS AND VALIDATION

A.1.1 DOMAIN AND STATE-SPACE SPECIFICATION

We begin with collected task examples and retain those that require querying or updating an executable environment, excluding requests answerable from static knowledge alone. For each retained example, we infer an application setting and specify its entities, fields, relations, and operational constraints. An order-management environment, for example, may contain customers, orders, products, and inventory. Each order refers to a customer, and its items refer to products. Rules can prohibit cancellation after shipment and prevent inventory from becoming negative. We then expand the state space beyond the initial examples so that the same environment can support additional workflows and tasks.

The generator implements each application setting as a Python environment with typed state containers and an initialization interface. The schema records integrity constraints, cross-object references, and business rules. These constraints govern both initial-state construction and subsequent tool execution.

A.1.2 TOOL CONSTRUCTION

Each tool specifies an exposed name, typed arguments, a return schema, a read or write designation, permitted state effects, and failure conditions. A refinement pass removes duplicate operations and decomposes overly broad ones into interfaces supporting multistep interaction. The generated methods are assembled with the environment class and checked for parseability, import consistency, and return structure.

A.1.3 EXECUTION CHECKS AND REPAIR

Validation first checks initialization, then executes tool sequences while recording pre-states, calls, observations, and post-states. Reads must leave state unchanged, and writes must respect their declared scopes, including consequential updates specified by their contracts. The checks also cover integrity constraints and observation structure. Invalid parameters and unmet business preconditions

must produce explicit failures. The resulting state is checked against the failure contract, which need not prescribe an unchanged state.

Local repairs are followed by revalidation of the affected behavior. Environments with unresolved failures are discarded. Each retained environment packages its application description, schema, initializer, tool schemas, executable transitions, and comparison policy. These checks establish consistency on the tested interactions.

A.2 Task Construction and Specification Audit

A.2.1 Workflow Proposals and Initialization

For each environment, we construct the tool dependency graph G_E described in Section 4.1. An edge indicates that one tool supplies an argument, identifies an entity, or establishes a state precondition for another. For example, searching for an order can identify its ID, and reading its status can establish whether cancellation is allowed. Before sampling, we remove unknown tool identifiers, self-loops, duplicate edges, and dependencies unsupported by the interfaces or environment rules.

Sampling favors source tools and tools with many outgoing dependencies. Bounded depth-first search collects longer chains, while random walks add structural variety. Search bounds limit chain length, and higher-difficulty tasks use longer chains than lower tasks. The accepted data have an average chain length of 69 tool calls. Each candidate must include at least one state-changing operation. A main chain can be supplemented with lookup, verification, or other relevant tools. We filter candidates that overlap excessively in tool nodes, dependency edges, or chain signatures, both within and across workflow groups.

For each candidate, the state generator instantiates the required objects, relations, and prerequisites. Initialization checks reject missing containers, invalid types or enum values, and broken references. Construction controls vary dependency length, distractor density, eligibility constraints, and evidence requirements, with any state augmentation validated before reference execution.

A.2.2 Grounded Reference Execution

Reference calls follow the task-specific dependency order. Arguments are grounded in preceding observations, explicit workflow parameters, or the initialized state. Before execution, arguments outside the exposed schema are removed and required parameters are checked. A failed call may receive local argument repair using the tool schema, error observation, and preceding execution context. An unresolved failure invalidates the candidate.

The completed execution record and terminal state S_g are retained for target extraction and task-description generation. Candidates with an empty net state change are excluded from the state-changing training set.

A.2.3 Task Description and Bidirectional Audit

The task generator uses the target state change, initial state, and execution record to express the required outcome and ground entity descriptions and constraints. The audit follows the coverage, justification, and grounding criteria in Section 4.1, with particular attention to two cases. First, an indirect entity description, such as a qualifying property, must identify the same object selected by the target using information available in the task description or exposed tools. Second, bookkeeping changes must be necessary consequences of the requested operations under the environment rules. Their occurrence in the reference record alone does not justify them.

Missing required effects, unsupported target effects, ambiguous references, and irreproducible targets trigger rejection or reconstruction followed by a new audit. The accepted description must specify the selected outcome sufficiently to fit the single-target formulation in Section 3, while leaving the solution path open.

A.2.4 Task Export and Scope

Alongside the task tuple $x = (E, q^\star, S_0, \Delta)$, the export stores S_g, the reference record, and the comparison policy for verification and auditing. The initial state, target, and comparison policy

remain fixed during subsequent trajectory synthesis and training, with agent-visible inputs restricted as specified in Section 4.2. State-change matching evaluates execution outcomes against this audited specification. Its reliability therefore depends on the specification's correctness.

B State-Change Verification and Reward Computation

This appendix makes the state-change verification and rollout-level reward in Sections 3 and 4.3 explicit. Given the initial state S_0, reference terminal state S_g, and rollout terminal state S_τ, the evaluator first identifies changes relative to S_0, then compares the changed values, and finally computes a state-change reward and an auxiliary format reward.

B.1 Changed Paths and Matching

The evaluator computes $\mathcal{P}_x = \mathrm{DiffPaths}(S_0, S_g)$ and $\mathcal{P}_\tau = \mathrm{DiffPaths}(S_0, S_\tau)$ separately. Membership in $\mathcal{P}_\tau$ establishes that a field actually changed; comparison with S_g then establishes whether the new value is correct. This order matters when a rollout leaves a required field at its initial value.

Changed-path extraction. Extraction visits the union of dictionary keys and recurses over keys present on both sides. A scalar or type change contributes its current path; an added or deleted key contributes its entire subtree as one path. A changed list also contributes one path, regardless of how many elements differ. No descendant paths are added beneath an atomic subtree change. Absence is represented explicitly and is distinct from a present null value. Fields unchanged between the two states contribute no path.

Recursive value comparison. For dictionaries, Equal_E requires identical key sets and recursively equal values, irrespective of serialization order. Lists use the multiset comparison in Appendix B.2; ordinary scalars are compared directly under the fixed comparison policy. An added dictionary is checked in its entirety: an extra field prevents a match even though the insertion contributes only one path. Likewise, replacing a present value with null is different from deleting its key.

Target-path matching. After the generated-record alignment in Appendix B.2, both changed-path sets are expressed in the target path space. A target path matches only when the rollout changes it and its terminal value agrees with the target:

$$\mathcal{P}_{\mathrm{match}} = \{p \in \mathcal{P}_x \cap \mathcal{P}_\tau : \mathrm{Equal}_E(S_\tau[p], S_g[p])\}. \tag{1}$$

A required deletion matches only actual absence. A target path absent from $\mathcal{P}_\tau$ is a missing change; a path present in both sets but with a different resulting value is an incorrect change. Both contribute zero to the matched-path count, while the required path remains in the reward denominator. Deleting a parent instead of updating a required child produces an unexpected parent path and leaves the child unmatched.

Unexpected-change weighting. The logical leaf count $L_p(S)$ is zero for absence, one for a scalar, an empty dictionary, or a list, and the sum of child counts for a nonempty dictionary. The penalty in Section 4.3 is computed as

$$u(p) = \max\{1, L_p(S_0), L_p(S_\tau)\}, \qquad U = \sum_{p \in \mathcal{P}_\tau \setminus \mathcal{P}_x} u(p). \tag{2}$$

Taking the maximum over the initial and terminal subtrees accounts for both deletions and additions. An unexpected dictionary is weighted by its logical leaf count, while a list remains one comparison unit regardless of length. The lower bound of one gives every unexpected path a positive penalty. An incorrect value on a required path is already counted as an unmatched requirement; it is not also placed in $\mathcal{P}_\tau \setminus \mathcal{P}_x$.

B.2 Value Comparison and New-Record Alignment

Path membership alone does not establish a match: the evaluator must compare complete changed values and identify corresponding newly created records when their keys are generated at execution time. The following list-comparison and record-alignment rules resolve these cases before the matched-path set in Equation 1 is computed.

Unordered lists. Lists are compared as multisets, retaining multiplicities and nested content. For equal-length lists $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_n)$, define

$$c_{ij} = \mathbf{1}[\neg\,\mathrm{Equal}_E(a_i, b_j)], \qquad C(A, B) = \min_{\pi \in \mathfrak{S}_n} \sum_{i=1}^{n} c_{i,\pi(i)}. \tag{3}$$

Here, $\mathfrak{S}_n$ is the set of permutations of n elements. A local Hungarian solver computes the minimum-cost one-to-one assignment; lists are equal exactly when $C(A, B) = 0$. Element comparisons recurse into dictionaries and nested lists without deduplication or requiring sortable elements. Unequal lengths cannot match, and two empty lists match.

Reordering alone creates no changed path. The assignment determines equality, not an element-level reward: a changed list receives credit only when the entire list matches. Order-sensitive requirements must encode order explicitly.

Generated record identifiers. When a tool creates a record, the environment may assign its key at execution time rather than taking it from the task description. The reference execution and the rollout can therefore assign different keys to records with the same task-relevant contents. A literal path comparison would then count the target addition as missing and the rollout addition as unexpected despite their equivalence. To avoid this, two dictionary records under different keys may align only if both are new relative to S_0 and share the same parent dictionary. For comparison, each generated key and string values within its record that exactly equal that key are replaced by `<generated-record-id>`. The complete remaining contents must agree recursively under the fixed comparison policy. Existing records and task-specified identifiers are excluded from this alignment.

Alignment is one to one: each rollout addition can satisfy at most one target addition. Matched rollout keys are mapped into the target path space before computing $\mathcal{P}_{\mathrm{match}}$ and $\mathcal{P}_{\mathrm{unexpected}}$; an unpaired target addition remains missing and an unpaired rollout addition remains unexpected. This local rule does not rename references across related records, which would require a consistent schema-level identity mapping.

B.3 State-Change Reward and Exact Matching

The state-change reward is computed once from the terminal state of a rollout. The evaluator counts the required paths $N = |\mathcal{P}_x|$, counts the required paths both changed and correct $M = |\mathcal{P}_{\mathrm{match}}|$, and sums the weights of rollout-only paths into U. For a state-changing task ($N > 0$), the reward in Section 4.3 is

$$R_{\mathrm{goal}} = \frac{|\mathcal{P}_{\mathrm{match}}|}{|\mathcal{P}_x| + U} = \frac{M}{N + U}. \tag{4}$$

Each matched requirement adds one to the numerator. A missing or incorrect requirement adds nothing but remains among the N required paths in the denominator; unexpected changes enlarge the denominator by U.

Exact state-change matching property. For $N > 0$, we have $0 \le M \le N$ and $U \ge 0$, giving

$$0 \le \frac{M}{N + U} \le 1, \qquad R_{\mathrm{goal}} = 1 \iff (M = N \text{ and } U = 0) \iff \Delta_\tau \equiv_E \Delta. \tag{5}$$

The first equivalence follows from $M = N + U$: with $M \le N$ and $U \ge 0$, equality requires $M = N$ and $U = 0$. For the second, $M = N$ means that every target path has changed to its required value, while $U = 0$ excludes any unexpected path because each has positive weight. Together these conditions give the exact-matching criterion in Section 3, under the same normalization and alignment rules.

Partial credit. For a fixed task and U, matching one additional target path increases the reward by $1/(N + U)$; for fixed $M > 0$, increasing U lowers the reward. The partial credit is per changed path: a partly correct list or a partly correct added record does not receive fractional credit within its single path.

B.4 Auxiliary Format Reward

The auxiliary format score uses the recorded action sequence and execution diagnostics. Let N_{attempt} count structured actions other than the final-response action, including structured calls later marked invalid. Let N_{valid} count those attempts with neither a parse-error nor an invalid-action flag. Their ratio is

$$r_{\text{call}} = \begin{cases} N_{\text{valid}}/N_{\text{attempt}}, & N_{\text{attempt}} > 0, \\ 0, & N_{\text{attempt}} = 0. \end{cases} \tag{6}$$

A raw action that fails parsing is not a structured attempt, but it still contributes to the parse-error count. Thus a rollout with no structured tool call gets no positive call-ratio credit even if it produces a final response. The auxiliary format reward is

$$\begin{aligned} R_{\text{format}} = \text{clip}_{[0,1]} \big(& r_{\text{call}} - \lambda_p n_p - \lambda_i n_i - \lambda_t I_t \\ & - \lambda_n I_n - \lambda_f I_f \big), \end{aligned} \tag{7}$$

Here n_p and n_i are the total parse-error and invalid-action counts; I_t, I_n, and I_f indicate a truncated rollout, failure to enter the environment's terminated state, and failure of the final-response check, respectively. All coefficients are nonnegative. A structured invalid action reduces r_{call} by remaining in N_{attempt} but not N_{valid}, and also contributes to n_i. This accounts separately for the reduced valid-call fraction and the explicit invalid-action penalty.

The final-response check requires the last structured action to be the designated communication action and its content, after trailing whitespace is removed, to end with the case-sensitive marker `Task Completed`. This marker affects I_f; it does not by itself establish task completion. A structurally valid tool call whose returned business result is unsuccessful can still count toward N_{valid}, while the resulting task outcome is measured through R_{goal}. Benchmark evaluation follows each benchmark's termination protocol.

B.5 Policy Optimization and Reward Interpretation

For RL, each rollout receives the complete task description $q^\star$ and starts from an isolated copy of S_0. After the rollout ends, its terminal state and diagnostics yield exactly the two scores defined above. The scalar used for policy optimization is

$$R = 0.9R_{\text{goal}} + 0.1R_{\text{format}}. \tag{8}$$

Worked calculation. Suppose a task has two required paths. A rollout matches one, leaves the other unchanged, and adds an unrelated dictionary with two scalar leaves. Then $M = 1$, $N = 2$, $U = 2$, and $R_{\text{goal}} = 1/4$. Suppose the same rollout attempts five structured non-final calls, four without error flags and one invalid, with all other penalty indicators zero. It has $r_{\text{call}} = 4/5$, $R_{\text{format}} = \text{clip}_{[0,1]}(4/5 - \lambda_i)$, and $R = 0.9(1/4) + 0.1\,\text{clip}_{[0,1]}(4/5 - \lambda_i)$. Thus outcome credit comes from terminal-state comparison, while format credit comes from recorded action diagnostics.

For each task, the rollouts in a group are compared using these terminal scalar rewards to form relative advantages. The policy loss is applied to model-generated responses and excludes user and tool messages. Any KL regularization belongs to the optimization objective rather than the reward in Equation 8.

Equation 5 concerns R_{goal}. Because Equation 8 also includes R_{format}, a higher total reward need not imply exact state-change matching. SFT acceptance continues to use the exact state-change criterion in Section 4.2.

C Trajectory Case Studies, Validation, and SFT Conversion

We illustrate Direct, Progressive, and Recovery through three trajectories in a synthetic hotel-audit task. Each panel preserves the sequence and arguments of the selected tool calls while abbreviating long messages and observations. Numbers identify steps in the trajectories, and ranges group separate calls.

C.1 DIRECT: GROUNDING AND EXECUTING A FULLY DISCLOSED REQUEST

The initial user message contains the full audit request, including an instruction to identify the top-ranked Amsterdam hotel, check the amenity catalog, and remove and restore a verified hotel–amenity association. The agent must first resolve the request to concrete entities before performing the prescribed update.

Direct: identifying the target hotel

User [0] (paraphrased). Audit customer `USR004`'s historical hotel searches; inspect both rankings and related records; reconcile one verified amenity of the top-ranked Amsterdam hotel by removing and restoring it after checking the full catalog.

Assistant → tool [2]. `get_user_search_history(user_id="USR004")`. **Tool:** search IDs `SQ001` and `SQ009`.
Assistant → tool [4, 6]. `get_search_query_by_id(query_id="SQ009")`, then `get_city_by_id(city_id="CTY009")`. **Tool:** `SQ009` belongs to Amsterdam.
Assistant → tool [8, 13]. `get_ranking_results_for_query(query_id="SQ009")`, then `get_hotel_by_id(hotel_id="HTL009")`. **Tool:** `HTL009` is rank 1; its name is Hotel Okura Amsterdam.

Omitted: other audit calls between the shown steps.

Direct: verifying and reconciling the amenity association

Assistant → tool [30]. `list_all_amenities()`. **Tool:** catalog contains `AMN001` (Wi-Fi).
Assistant → tool [31]. `list_amenities_for_hotel(hotel_id="HTL009")`. **Tool:** `AMN001` is associated with the hotel.
Assistant → tool [56]. `get_user_by_id(user_id="USR001")`. **Tool:** active administrator.

Omitted: other audit calls at steps 32–55 and 57–65.

Assistant → tool [66]. `remove_amenity_from_hotel(hotel_id="HTL009", amenity_id="AMN001", user_id="USR001")`. **Tool:** removal succeeds.
Assistant → tool [67]. `add_amenity_to_hotel(amenity_id="AMN001", current_user_id="USR001", hotel_id="HTL009")`. **Tool:** association succeeds.
Assistant → tool [68]. `list_amenities_for_hotel(hotel_id="HTL009")`. **Tool:** `AMN001` appears again among the hotel's six amenities.

Complete-goal planning. The agent grounds the initially specified audit request by resolving the customer search, city, ranking result, hotel, catalog entry, and existing association before acting. It also checks an administrator account before the write calls. The remove–restore sequence is specified from the outset, and the final read verifies the local association. This case demonstrates entity grounding and execution under complete initial disclosure.

C.2 PROGRESSIVE: A NEW REQUEST AFTER COMPLETED WRITES

The second trajectory begins with a request to retrieve the customer's search history. Later user turns provide operational details. Steps 84–97 show the transition from review voting to booking cleanup and the preservation of earlier effects.

Progressive: completing the review request

User [0] (paraphrased). Start the audit for `USR004` by retrieving the customer's hotel search history and identifying two searches.

Intermediate audit and clarification steps omitted.

Assistant → user [84] (paraphrased). Requests the acting user and review IDs for helpfulness votes; reports that flags `ML001` and `ML005` should remain.
User [84] (paraphrased). Vote as `USR004` on `REV002`, `REV003`, `REV024`, `REV019`, `REV020`, and `REV035`; leave both flags in place.
Assistant → tool [85–90]. Six separate `vote_review_helpful` calls, one for each listed `review_id`, all with `user_id="USR004"`. **Tool:** all six calls report success.
Assistant → tool [91]. `get_review_by_id(review_id="REV035")`. **Tool:** `helpfulness_votes=10`; an earlier read had returned 9.

Progressive: the subsequent booking request

Assistant → user [92] (paraphrased). Reports the six votes and preservation of `ML001` and `ML005`.
User [92] (paraphrased). Now cancel the discovered pending bookings that belong to their booking owners and are suitable for cleanup.

Assistant → tool [93–94]. `get_booking_by_id(booking_id="BKG002")` and `get_booking_by_id(booking_id="BKG018")`. **Tool:** both are `pending`, owned by `USR005`, with empty `payment_id`.
Assistant → tool [95–96]. Two `cancel_booking` calls, for `BKG002` and `BKG018`, each with `user_id="USR005"`. **Tool:** both succeed.
Assistant → tool [97]. `get_booking_by_id(booking_id="BKG002")`. **Tool:** `status="cancelled"`.

Progressive: cumulative state changes

Review vote counts, prior reads → terminal state: `REV002` 8 → 9; `REV003` 15 → 16; `REV024` 19 → 20; `REV019` 8 → 9; `REV020` 13 → 14; `REV035` 9 → 10. The prior values appear in steps 12, 13, 69, and 70.
Booking status, prior reads → terminal state: `BKG002` and `BKG018` both `pending`→`cancelled`. Moderation flags `ML001` and `ML005` remain in place.

Cumulative effects across user requests. The booking request follows the assistant's review-completion message at step 92. The agent checks both bookings before canceling them under their recorded owner, `USR005`, while the terminal state retains the earlier vote increments. The case thus exhibits successive disclosure and accumulation of the listed effects.

C.3 Recovery: Repairing a Disrupted Amenity Operation

To construct the Recovery trajectory, we manually introduce a disruption at the `HTL009`–`AMN001` amenity-removal step. The first two removal attempts receive permission-denied responses, interrupting the planned remove–restore sequence. The agent checks account roles, changes the acting user, and completes the original request.

Recovery: controller-induced authorization disruption

Assistant → tool [38]. `list_amenities_for_hotel(hotel_id="HTL009")`. **Tool:** `AMN001` is present.
Assistant → tool [39]. `list_all_amenities()`. **Tool:** catalog contains `AMN001` (Wi-Fi).

Controller [40]. Before the next agent action, the controller perturbs the authorization state by changing `USR004`'s role from `admin` to `customer`. The intervention is not directly disclosed to the agent.
Assistant → tool [41]. `remove_amenity_from_hotel(hotel_id="HTL009", amenity_id="AMN001", user_id="USR004")`. **Tool:** failure; `USR004` now has role `customer` and lacks permission.
Assistant → tool [42]. `get_user_by_id(user_id="USR002")`. **Tool:** active moderator.
Assistant → tool [43]. `remove_amenity_from_hotel(hotel_id="HTL009", amenity_id="AMN001", user_id="USR002")`. **Tool:** failure; the moderator also lacks permission.
Assistant → tool [44]. `get_user_by_id(user_id="USR001")`. **Tool:** `role="admin", status="active"`.

Recovery: revised action and state restoration

Assistant → tool [45]. `remove_amenity_from_hotel(hotel_id="HTL009", amenity_id="AMN001", user_id="USR001")`. **Tool:** removal succeeds.
Assistant → tool [46]. `add_amenity_to_hotel(hotel_id="HTL009", amenity_id="AMN001", current_user_id="USR001")`. **Tool:** association succeeds.

Controller [47]. After the recovery sequence, the controller restores the perturbed authorization state by changing `USR004`'s role from `customer` back to `admin`.

Omitted: other audit calls at steps 48–62.

Assistant → tool [63]. `list_amenities_for_hotel(hotel_id="HTL009")`. **Tool:** `AMN001` is again present.
Assistant → tool [64]. `get_user_by_id(user_id="USR004")`. **Tool:** `role="admin"`, confirming that the controller-induced authorization change has been reverted.

Recovery after disruption. The controller-induced intervention temporarily changes the authorization state of the user initially selected for the removal, causing the next tool call to fail. The agent observes the permission failure, checks alternative user roles, unsuccessfully tries an active moderator, and then identifies the active administrator `USR001`. Reissuing the removal with valid credentials succeeds, after which the requested amenity association is restored. The controller then reverts its temporary authorization perturbation, and the final reads verify both the requested amenity state and restoration of the unrelated user state. The trajectory therefore realizes the disruption–observation–revision–recovery–verification pattern while leaving no residual controller-induced state change.

C.4 Trajectory Construction and SFT Conversion

For the synthesis procedure in Section 4.2, the runner records user messages, assistant actions, tool observations, state snapshots, termination status, and condition-specific controller events for replay.

C.4.1 Condition Eligibility and Controller Events

Direct. The complete task description is placed in the initial user message, and the designated communication action ends the interaction.

Progressive. Stage targets use strict inclusion of required path–value effects under the fixed comparison policy. Each stage must be reachable from the preceding cumulative state through the exposed tools, including any consequential updates. For example, if the first stage fixes a counter to one and the second necessarily increments it to two, the stages cannot form the required nested targets. The same restriction applies to an atomic list whose required value would change between stages. Such effects must be grouped into a compatible stage rather than omitted from comparison.

Each reveal record stores the stage index, state snapshot, and gate result. After the current gate passes, the newly revealed description is appended to the interaction history; future descriptions and

cumulative targets remain hidden. Clarifications and read-only exchanges may occur within a stage without defining an additional state-change target.

Recovery. An intervention specifies a trigger, a state transformation, and the resulting agent-visible observation, and is applied at a declared boundary between agent actions. It disrupts a completed update or a prerequisite while satisfying the invariant and reachability requirements in Section 4.2.

At the actual trigger state, a recovery witness is executed on an isolated perturbed-state copy to verify that the full target Δ remains achievable relative to S_0. The witness is retained for auditing but is not exposed to the agent or executed on the live trajectory. The event record stores the trigger outcome, pre- and post-intervention states, agent-visible observation, and reachability-check result. A rollout is retained as a Recovery candidate only if an intervention was applied.

This check includes consequential effects of the repair. For example, restoring a deleted association is insufficient if the repair necessarily leaves an additional audit record outside Δ. In that case, the intervention does not satisfy the reachability requirement even though the association itself can be restored.

C.4.2 Replay and Acceptance

Message processing preserves call–response associations and rejects unsupported action structures. Changes to the executable history require replay from S_0, including recorded disclosures and interventions, with their preconditions and observations checked for consistency.

Record and protocol checks establish whether a trajectory is a valid execution under the specified interaction condition. Progressive records must complete disclosure gates in order, and Recovery records must contain an applied admissible intervention. Applicable response obligations and interaction constraints are checked against the recorded history. Records with failed or undefined replay are excluded.

For eligible candidates, the verifier computes $\mathrm{Diff}(S_0, S_\tau^\kappa)$ from the replayed terminal state and applies $\mathrm{Accept}_x(\tau, \kappa)$ from Section 4.2. This keeps record and protocol checks separate from the outcome criterion. A temporary execution error need not cause rejection if its state effects are repaired and the interaction requirements remain satisfied.

C.4.3 Representative Selection

For each task and condition, we retain the most efficient accepted trajectory, as specified in Section 4.2. A singleton pool contributes its sole candidate; an empty pool contributes none. Selection therefore produces at most one retained trajectory per task and condition before conversion into turn-level training records.

C.4.4 Information-Preserving SFT Conversion

The converter applies the target model's chat template to structured messages and tools. For a retained assistant turn a_t, the training example pairs its preceding interaction history H_t with a_t as the supervised output. Loss is applied only to a_t; system and user messages, tool observations, and earlier assistant turns provide context without contributing loss in that example.

For successive assistant turns a_t and a_{t+1} separated by a tool observation o_t, the next example uses $H_{t+1} = (H_t, a_t, o_t)$ and supervises only a_{t+1}. Newly disclosed requirements or perturbation observations enter the history at their recorded positions. Controller-only targets, validation witnesses, and undisclosed requirements are excluded. Thus, one retained trajectory yields multiple SFT records, each preserving the information available at its supervised turn.

If tool or parameter identifiers are anonymized, the same mapping is applied to schemas, calls, and associated messages while preserving argument meaning and call–response identity. Original execution records remain available for replay.

D Experimental Details and Supplementary Results

This appendix provides the SFT configuration, clarifies how training records and evaluation aggregates are counted, and reports complete training-data comparison scores, Shopping results by difficulty level, and training-stage differences.

D.1 Training-Record Granularity

StateForge counts each supervised assistant turn as one SFT record, following the conversion in Appendix C.4.4. The recorded EnvScaler and EnvFactory inventories contain approximately 9K and 26.5K SFT records, respectively (Song et al., 2026; Xu et al., 2026a). These counts do not establish equal token exposure: record granularity and interaction length may differ across sources, and the shared epoch count does not match supervised tokens or optimizer updates.

D.2 SFT Configuration

Table 4 gives the Qwen3.5-4B SFT configuration used for all three data sources in Section 5.3. StateForge-9B shares the listed optimizer, learning rate, schedule, warmup ratio, epoch count, sequence length, GPU count, and gradient accumulation.

Table 4: Qwen3.5-4B SFT configuration, shared across all three data sources in the training-data comparison.

Setting	Value
Training framework	LLaMA-Factory
Fine-tuning method	Full-parameter language fine-tuning; vision tower and multimodal projector frozen
Optimizer	PyTorch AdamW
Learning rate	1×10^{-6}
Training epochs	3
Maximum sequence length	32k tokens
Number of GPUs	8
Per-device batch size	1
Gradient accumulation	32 steps
Effective batch size	256 training examples per optimizer update
Learning-rate schedule	Cosine decay
Warmup ratio	0.1
Maximum gradient norm	1.0
Chat template	`qwen3_5`, thinking enabled
History masking	Enabled
Sequence packing	Disabled
Numerical precision	bfloat16
Memory and attention	Gradient checkpointing, FlashAttention-2, DeepSpeed

D.3 Evaluation Aggregation

Shopping Acc. measures complete-cart correctness, while Match Score credits partial product matching under the DeepPlanning evaluator (Zhang et al., 2026). We use the evaluator's aggregate Acc. directly; the displayed L1–L3 scores are separate breakdowns and are not used to reconstruct it. The BFCL-MT Overall column averages only the four multi-turn categories named in Section 5.1, so it should not be read as the full BFCL-v3 leaderboard score.

D.4 Complete Training-Data Comparison Results

Table 5 reports the complete absolute scores underlying Figure 3. All three training corpora are evaluated using the same Qwen3.5-4B backbone and SFT configuration. Base denotes the backbone before fine-tuning.

Table 5: **Complete training-data comparison on Qwen3.5-4B at the SFT stage** (%, ↑). Absolute scores corresponding to Figure 3. Base denotes the backbone before fine-tuning. Bold indicates the best fine-tuned result in each column.

Training data	Shopping		τ^2-Bench			
	Acc.	Match	Airline	Retail	Telecom	Macro
Base	35.83	74.36	76.00	77.20	100.00	84.40
EnvFactory data	11.67	46.52	60.00	77.19	85.09	74.09
EnvScaler data	15.00	56.04	66.00	68.42	71.39	68.60
StateForge data	**42.50**	**78.57**	**78.00**	**78.90**	**99.10**	**85.33**

D.5 Shopping Results by Difficulty Level

Table 6 breaks down the Shopping evaluations in Table 1 by constraint level, omitting models without reported level scores.

Table 6: Shopping accuracy by difficulty level (%). Aggregate Acc. and Match scores appear in Table 1. A dash in Stage indicates an external baseline.

Model	Stage	L1	L2	L3
AWM-14B	—	16.00	6.00	0.00
EnvScaler-8B	—	16.00	10.00	0.00
EnvFactory-8B	—	8.00	2.00	0.00
AgentSkiller-14B	—	14.00	8.00	5.00
TOUCAN-32B	—	16.00	4.00	0.00
Qwen3.5-4B	Base	40.00	30.00	40.00
StateForge-4B-SFT	SFT	54.00	28.00	50.00
StateForge-4B-RL	RL	54.00	32.00	45.00
Qwen3.5-9B	Base	46.00	36.00	40.00
StateForge-9B-SFT	SFT	52.00	36.00	55.00
StateForge-9B-RL	RL	56.00	38.00	45.00

At both sizes, SFT improves L1 and L3 over Base, but L2 falls at 4B and remains unchanged at 9B. Continued RL improves L2 and reduces L3 at both sizes; L1 remains unchanged at 4B and improves at 9B. Thus, the aggregate gains in Table 1 do not imply gains at every constraint level.

Table 7 further breaks down the Shopping results from Table 5 by difficulty level.

Table 7: Shopping level accuracy for the training-data comparison on Qwen3.5-4B at the SFT stage (%). Base denotes the backbone before fine-tuning.

Training data	L1	L2	L3
Base	40.00	30.00	40.00
EnvFactory	20.00	8.00	0.00
EnvScaler	18.00	14.00	10.00
StateForge	54.00	28.00	50.00

StateForge exceeds both alternative training corpora at each level. Relative to Base, it improves L1 and L3, while L2 decreases by 2 pp.

D.6 Training-Stage Decomposition

Table 8 separates SFT gains, additional RL gains, and total gains over Base for the checkpoints in Table 1.

At 4B, RL increases Shopping Acc. by 0.83 pp while reducing Match Score by 0.73 pp. The net τ^2 Macro gains over Base come from Airline and Retail, since Telecom is at 100% for both Base and RL checkpoints at each size.

Table 8: Training-stage differences (pp), computed before rounding. Positive values indicate improvement. SFT and RL denote the corresponding StateForge checkpoint suffixes; Base denotes the Qwen3.5 backbone of the same size.

Size	Comparison	Shopping Acc.	Match	τ^2 Macro	BFCL-MT
4B	SFT minus Base	6.67	4.21	0.93	1.75
4B	RL minus SFT	0.83	-0.73	5.83	0.25
4B	RL minus Base	7.50	3.48	6.77	2.00
9B	SFT minus Base	5.00	0.74	1.64	1.63
9B	RL minus SFT	0.84	1.82	1.17	0.13
9B	RL minus Base	5.84	2.56	2.81	1.75