ENVCRAFT: SYNTHESIZING EXECUTABLE ENVIRONMENTS IN AGENTIC RL FOR CLAW-LIKE AGENT

Anonymous authors
Paper under double-blind review

ABSTRACT

The paradigm of LLMs has rapidly shifted from passive language interfaces to autonomous Claw-like agents that execute long-horizon tasks across stateful workspaces. While Agentic Reinforcement Learning (Agentic RL) provides a promising path to optimize these agents, its scaling is heavily bottlenecked by the severe scarcity of interactive training environments. Existing synthetic environments are strictly limited to tool-calling endpoints, rendering them insufficient for accommodating the end-to-end real-world demands of claw-like agents. To bridge this gap, we introduce EnvCraft, an automated framework for synthesizing executable environments and scalable training data. Specifically, EnvCraft employs an environment synthesis engine to build sandbox-isolated workspaces, alongside a topology-aware data generation engine to produce coherent task specifications. Overall, we synthesize 139 interactive environments comprising approximately 20K complex tasks for Agentic RL training. Experiments on Qwen3/3.5 models (8B–32B) show that our method yields gains of up to +11.9 points on Claw-style benchmarks and +8.0 points on general tool-use benchmarks, with concurrent reductions in inference token cost. Taken together, these results support automated, large-scale synthesis of executable environments as a practical foundation for training Claw-like agents.

1 INTRODUCTION

AI agents have evolved from passive conversational interfaces into autonomous actors operating in real-world systems. A notable embodiment of this evolution is the class of claw-like agents, designed for efficient execution within system-level harnesses. These harnesses, such as OpenClaw (OpenClaw Contributors, 2026), NanoClaw (QwibitAI, 2026), and Hermes-Agent (Nous Research, 2026), function as persistent digital assistants that manage long-horizon tasks across operating systems, file systems, databases, and terminal shells. Unlike traditional conversational bots, claw-like agents possess active execution capabilities that directly mutate the environment, producing real-world consequences. Recent works explore *Agentic RL*, which enables agents to iteratively refine their policies, recover from execution errors, and generalize to unseen scenarios.

Agentic RL frameworks inherently rely on the interaction among users, agents, and environments, as shown in Figure 1. The environment is an executable sandbox that supports tool execution, state maintenance, and result feedback, while an executable environment is foundational to this paradigm, scaling end-to-end Agentic RL training critically requires environments to be low-latency, high-fidelity, and cost-effective. Currently, environments meeting these rigorous criteria remain scarce, limited primarily to narrow domains such as code interpreters (Feng et al., 2025) and local-retrieval tools (Jin et al., 2025). This scarcity becomes particularly acute for Claw-like agents, which demand stateful workspaces and long-horizon task loops, requirements that far exceed the complexity of tool-use endpoints. Consequently, the lack of scalable, interactive, and executable environments stands as the key bottleneck for advancing Agentic RL at the system level.

To address this bottleneck, recent studies have explored two main paradigms for generalist environment construction: *simulated environments*, which leverage LLMs to mock tool responses (Chen et al., 2025b; Li et al., 2025c; 2026b) but suffer from hallucination and high costs; and *synthetic environments*, which auto-generate executable code sandboxes for better scalability (Song et al., 2026; Xu et al., 2026). However, contemporary synthetic methods focus on isolated tool-use endpoints at the

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

application level, and fail to support system-level *Claw-like agents*, which demand cross-environment execution, stateful workspaces, and asynchronous task loops. To bridge this critical gap, we present EnvCraft, an automated executable environment synthesis framework tailored specifically for Claw-like agent training while also supporting general-purpose tool-use agents. Table 1 summarizes the comparison results.

Specifically, **EnvCraft** operates through two complementary engines: environment synthesis and data generation. For environment synthesis, we curate 41 general tool-use and 42 Claw-specific scenarios, which LLMs expand into detailed environment specifications and tool interface definitions. Guided by nine interaction archetypes that prescribe control-flow and state-management patterns, the synthesizer generates fully executable, sandbox-isolated environments. For data generation, we introduce a topology-aware sampling strategy to produce high-quality RL tasks comprising user intents, initial states, and verifiers. We first build intra- and cross-environment tool dependencies as a bi-level directed graph, from which weighted random walks yield semantically coherent and logically valid tool chains. Conditioned on these chains, LLMs reverse-engineer multi-turn user intents that expose information incrementally, while simultaneously generating deterministic verification scripts that check post-execution workspace states to supply reward signals. Using **EnvCraft**, we synthesized 139 interactive environments and 19,777 tasks spanning both Claw-specific and general tool-use task.

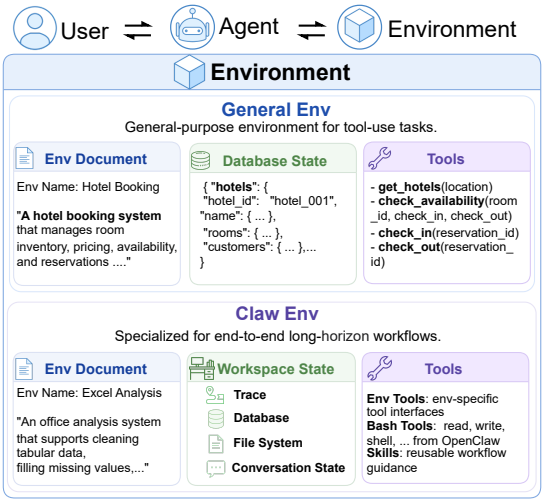


Figure 1: An illustration of the agent interaction loop in RL. Environment fundamentally comprises environment documentation, explicit state definitions, and a suite of tool interfaces.

Extensive experiments on Qwen3-8/32B and Qwen3.5-9B backbones provide empirical evidence for the training effectiveness of the EnvCraft pipeline as a whole. On Claw-style benchmarks, Claw-specific training yields gains of up to +11.9 points on PinchBench and +11.4 on Claw-Eval, while simultaneously reducing per-task token cost by up to 35%, which is consistent with more efficient reasoning behavior. These improvements are not confined to Claw-style evaluation: the same models also improve on general-purpose tool-use benchmarks (BFCL-v3, τ^2-bench), suggesting that training on synthesized Claw environments does not come at the cost of standard multi-turn tool-use ability. Taken together, these results position executable environment synthesis as a practical path to scaling Agentic RL at the system level: the environments, tasks, and verifiers that Claw-like training requires can be generated automatically at low cost, without hand-built sandboxes or human environment engineering.

Environment	# Envs	# Tasks	Claw?	Env Domain	Stateful	Reward
AutoForge(Cai et al., 2025)	10	1,078	✗	Consumer Web	DB Schema	LLM Judge
EnvScaler(Song et al., 2026)	191	9,000	✗	General	Class Attr.	Rule Check
ScaleEnv (Tu et al., 2026)	16	2,560	✗	General	DB Schema	Rule Check
AWM(Wang et al., 2026)	1,000	10,000	✗	General	DB Schema	Code & LLM
EnvFactory(Xu et al., 2026)	85	2,575	✗	General	DB Schema	Code Check
EnvCraft (Ours)	139	**19,777**	✓	General	Workspace	Code Check

Table 1: Comparison of synthetic executable environments for Agentic RL. **Claw?** indicates whether the environment supports Claw-like agent training, as opposed to isolated tool endpoints. Unlike existing works limited to generalist tool endpoints, EnvCraft provides stateful, workspace-level infrastructure for training *Claw-like agents* with the largest task scale.

2 PRELIMINARIES

2.1 PROBLEM SETUP: AGENTIC INTERACTION

We model an executable agentic environment E as a quadruple that bridges reinforcement learning abstractions with LLM-based interaction: $E = \langle \mathcal{D}_{\text{doc}}, \mathcal{I}_{\text{tool}}, \mathcal{S}, \mathcal{T} \rangle$ $\mathcal{D}_{\text{doc}}$ is a natural-language description of the global system rules, operational constraints, and safety boundaries; $\mathcal{I}_{\text{tool}}$ specifies the names, parameters, and descriptions of all tools exposed to the agent; $\mathcal{S}$ is the persistent, observable state space $\mathcal{S}_t = \{s_{\text{data}}, s_{\text{runtime}}\}$, where s_{data} captures underlying data states (*e.g.*, file metadata or database schemas) and s_{runtime} temporal execution sessions; and $\mathcal{T}$ executes an agent action $a_t \in \mathcal{I}_{\text{tool}}$ within an isolated sandbox, deterministically yielding the next state and observation, $\mathcal{T} : \mathcal{S}_t \times \mathcal{I}_{\text{tool}} \to \mathcal{S}_{t+1} \times \mathcal{O}_{t+1}$.

To facilitate optimization via Agentic RL, we formalize each training data point $\mathcal{P}$ generated within a synthetic environment as a structured tuple: $\mathcal{P} = \langle \mathcal{Q}, \mathcal{S}_0, \mathcal{V}_{\text{script}} \rangle$ where $\mathcal{Q}$ is the user query, $\mathcal{S}_0 \in \mathcal{S}$ the concrete initial state instantiating the runtime workspace prior to agent execution, and $\mathcal{V}_{\text{script}}$ an automated verification script that programmatically audits the state transitions $\mathcal{S}_0 \to \mathcal{S}_n$ (*e.g.*, verifying structural file mutations or executing exact metadata checks). This design yields deterministic, programmatic reward signals.

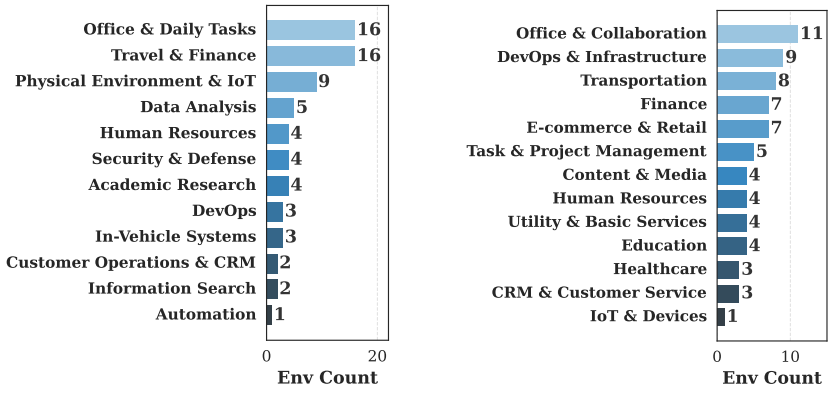
3 ENVIRONMENTAL SYNTHESIS ENGINES

Operating in an end-to-end fashion, the environment synthesis engine fully automates the transition from natural-language descriptions to executable environments.

3.1 ENV SCENARIO COLLECTION

We gathered environment domains spanning both general tool-use and Claw-specific scenarios, curated from the OpenClaw community (Steinberger, 2026; Tencent, 2026; Sheikh, 2026) and representative prior works (Patil et al., 2025; Chen et al., 2025a; Song et al., 2026). After deduplication and semantic filtering, we retained 12 coarse-grained general tool-use domains and 13 Claw-specific categories. At a finer granularity, these categories encompass 42 Claw-specific scenarios and 41 general tool-use scenarios, with full specifications in Sec. B.2.

We then use LLMs to expand these skeletal profiles into granular, document-level environment specifications ($\mathcal{E}_{\text{spec}}$) and declarative tool interface specifications ($\mathcal{T}_{\text{spec}}$).



(a) Claw-Specific Scenarios (b) General Tool-Use Scenarios

Figure 2: Distribution of synthesized environments across coarse-grained scenario categories.

3.2 ENVIRONMENT SANDBOX SYNTHESIS

We propose a prototype-driven synthesis framework that turns natural-language descriptions into executable sandboxes.

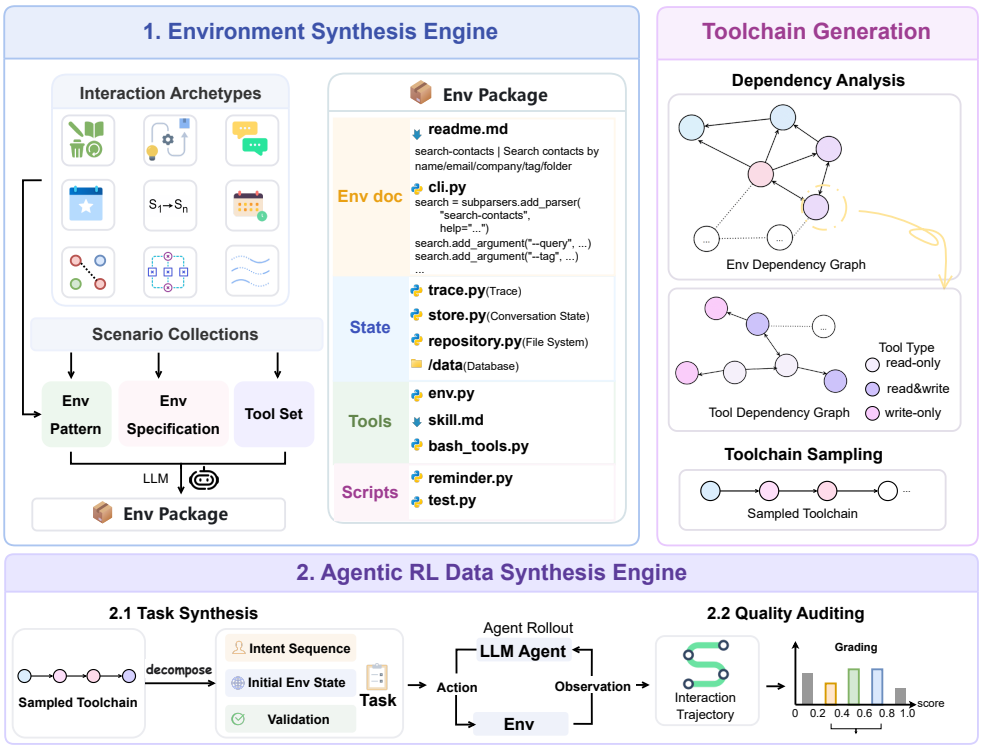


Figure 3: Environment and task synthesis pipeline in EnvCraft. Given natural-language descriptions, the Environment Synthesis Engine generates executable sandboxes (state + tools + docs); the Data Synthesis Engine samples tool chains and synthesizes RL training tasks (intent sequence + initial state + verifier).

Interaction Archetype Design and Matching. To accommodate the complex interaction requirements in OpenClaw scenarios, we distill software interaction archetypes into nine foundational *prototypes*. Each interaction archetype functions as a structured prompt template, combining abstract design specifications with concrete reference implementations and architectural blueprints. These nine prototypes cover distinct control-flow and state-management paradigms: (1) transactional workspace, (2) cross-tool data pipelines, (3) dynamic human-agent interaction, (4) asynchronous event triggering, (5) constraint-guided state machine, (6) multi-resource temporal scheduling, (7) multi-criteria bilateral matching, (8) distributed workflow orchestration, and (9) fault-tolerant stream processing. The complete interaction archetype specifications are provided in Sec. B.3.

To ground LLM generation, the framework retrieves relevant prototypes by scoring keyword overlap with the environment specification ($\mathcal{E}_{\text{spec}}$). The candidate prototypes are then ranked, and up to the top two are selected as in-context architectural anchors.

Specification-Guided Environment Generation. The engine ingests three structured inputs: the retrieved interaction archetype specifications ($\mathcal{P}_{\text{proto}}$), which supply concrete design prototypes and reference implementations; the environment specification ($\mathcal{E}_{\text{spec}}$), defining domain entity types, state attributes, and business rules governing valid state transitions; and the tool specification ($\mathcal{T}_{\text{spec}}$), formulating the API surface with function schemas, typed parameters, return formats, and side-effect flags (*e.g.*, read-only vs. write-only). A unified structured prompt integrates all three, and the LLM generates a fully executable environment class conforming to the resulting contract. A representative synthesized instance is illustrated in Figure 3; the system prompts and execution schemas are detailed in Sec. C.

Claw-Specific Extension. To accommodate the unique execution demands of Claw-specific scenarios, we introduce two domain-specific enhancements:

- **Built-in Tools:** We inject a system-level Bash execution tool derived from the OpenClaw harness into the environment's default toolset, granting agents direct CLI-level interaction capabilities.
- **Parallel Skill Co-Synthesis:** Concurrently with the environment generation, we prompt the synthesizer to produce a set of domain-tailored *Agent Skills* (reusable workflow guidance in markdown).

3.3 Environment Quality Auditing

All synthesized environments undergo a dual-stage automated quality control protocol.

(1) **Static code analysis**. AST parsing and predefined compliance rules verify that all tool schemas and interface definitions adhere to the structural specifications.

(2) **Dynamic runtime analysis**. We instantiate each environment from its initial state $\mathcal{S}_0$ and execute predefined test cases, validating state transition logic ($\mathcal{S}_t \rightarrow \mathcal{S}_{t+1}$) and auditing exception handling under edge cases. Environments passing both stages enter the final library; failures are routed to a self-healing loop in which an LLM patches the codebase from execution logs for re-verification.

Finally, as shown in Table 2, quality auditing reduced the 324 raw environments to 139 (42.9% pass rate), comprising 70 general-tool environments and 69 Claw-specific environments.

4 Data Synthesis Engines

4.1 Sampling Tool Chain

To ground subsequent task synthesis in valid execution logic, we model tool and environment dependencies as a bi-level directed graph structure. At the bottom level, we construct an intra-environment *tool dependency graph* for each sandbox. By combining static code analysis with LLM reasoning over tool pre- and post-conditions, we map out logical sequences (e.g., data flows) as weighted directed edges. At the top level, to support cross-system workflows, we build a cross-environment *env dependency graph*. Here, environments act as nodes, and the LLM establishes weighted edges based on semantic domains and cross-environment data flows.

Env/Data Types	Passed	Raw	Pass Rate
Environment			
General Tool Env	70	203	34.48%
Claw Env	69	121	57.02%
# Total	139	324	42.90%
Data Entry			
General Tool Data	14,215	21,975	64.69%
Claw-special Data	5,562	13,489	41.23%
# Total	19,777	35464	55.77%

Table 2: Statistics of Environment and Data from EnvCraft, along with pass rate in quality auditing.

We generate executable tool chains via a hierarchical weighted random walk over this bi-level structure. For single-environment tasks, the sampler traverses the tool dependency graph; for cross-environment tasks, it first samples a sequence of logically connected environments (typically 2–5) from the env-graph, then performs intra-environment walks within each tool-graph to generate and concatenate prefixed sub-chains. To ensure verifiability, we partition all tools into *read tools* (non-state-mutating) and *write tools* (state-altering), and enforce that every sampled chain terminates with a mutation tool. This guarantees that each synthesized task ultimately modifies the environment state, enabling programmatic answer verification through state-based evaluation.

4.2 Task Synthesis

We transform each sampled tool chain into a self-contained training sample in three stages.

Chain-to-Task Reverse Engineering. The LLM analyzes data-flow dependencies along the chain and classifies each parameter as either *user-specified* (grounded literals provided by the user, e.g., order IDs or file paths) or *tool-derived* (intermediate results resolved from preceding tool outputs, e.g., query-returned record IDs). It then reverse-engineers a natural-language user instruction in which user-specified values are stated explicitly while tool-derived values are referenced only abstractly,

never revealing their ground-truth values. This information-hiding mechanism forces the downstream agent to discover intermediate results through actual tool execution, ensuring task non-speculability.

Intent Decomposition Since the above intents are expressed as single composite instructions while real users disclose information incrementally, we decompose each task into a sequence. Given the composite intent and the tool chain's step structure, the LLM generates a user instruction sequence in which the user reveals information or poses sub-tasks step-by-step, and the agent executes corresponding tools sequentially until the overall objective is fulfilled.

To improve the naturalness of the user intents, we incorporate persona attributes into the decomposition process. Specifically, we sample persona profiles from PersonaHub (Ge et al., 2024) and compute their semantic similarity with the documentation of all environments. For each environment, we retain up to the five most relevant personas (500 in total), and randomly select one during intent decomposition to condition the simulated user's behavior.

Intent-to-Environment Construction. Given the complete environment and tool source code together with the decomposed intent sequence, this step generates the initial environment state and the corresponding answer verification script. We prompt an LLM to instantiate the initial state by populating the workspace with both the entities required for the correct execution path and plausible distractor records. In parallel, the LLM generates a deterministic verification script that checks whether the resulting workspace state satisfies the user's intent. The result is a self-contained JSON task record.

4.3 Data Quality Auditing

To ensure the fidelity and trainability of synthesized data, we apply an auditing protocol: First, we employ an LLM with a predefined rubric covering instruction clarity, tool chain consistency, and evaluation-function correctness; samples scoring below the acceptance threshold are discarded. Second, we perform difficulty calibration by running Qwen3.5-27B on each task for eight independent trials and filtering out samples with a 0% or 100% pass rate. Samples whose pass rates fall within an intermediate range, constitute the final training set with a balanced difficulty distribution (He et al., 2025).

After auditing, we retain 19,777 high-quality samples from 35k raw candidates (55% pass rate), comprising 14,215 general tool-use (**EnvCraft-Tool**) and 5,562 Claw-specific instances (**EnvCraft-Claw**), as shown in Table 2.

5 Experiments

5.1 Setup

Implementation. We fine-tune Qwen3 and Qwen3.5 series models (ranging from 8B to 32B parameters) using GRPO within the VERL framework (Sheng et al., 2024). Our primary training strategy focuses on claw-specialized data; we additionally explore a two-stage curriculum that first trains on general tool-use data before fine-tuning on Claw-specific trajectories. Training is conducted on 64 GPUs with a rollout batch size of 64, a maximum generation length of 32k tokens, and up to 64 action turns per trajectory. We enable the default thinking mode throughout training and inference, and defer remaining hyperparameters to Sec. D.1. Environment, task, and verifier synthesis is performed entirely with DeepSeek series models; producing all environments and tasks costs approximately $420 in API usage and requires no human environment engineering.

Baselines. We evaluate EnvCraft on the open-source Qwen3 and Qwen3.5 model series and compare them against representative frontier models, including GPT-5.4 (OpenAI, 2026), Claude Opus 4.6 (Anthropic, 2026), Kimi K2.5 (Moonshot AI, 2026), and MiniMax M2.7 (MiniMax, 2026). Additionally, we include recent RL data for general tool-use environment synthesis (see Table 1): EnvScaler (Song et al., 2026), ScaleEnv (Tu et al., 2026), AWM (Wang et al., 2026), and EnvFactory (Xu et al., 2026). These baselines were trained under heterogeneous recipes that we could not fully replicate, so we report the numbers published in their original papers rather than re-training each method under a matched budget; cells marked "–" denote settings that the corresponding original

Model	PinchBench		ClawEval	
	Score	Token	Score	Token
Advanced Models				
Claude Opus 4.6	69.90	12.30	80.60	20.40
Kimi-2.5	54.60	14.00	66.60	17.80
GPT 5.4	75.70	11.50	78.30	15.30
MiniMax-m2.7	65.40	11.60	71.80	17.80
Main Results				
Qwen3-8B	12.94±0.3	13.69	44.06±0.2	15.49
+ EnvCraft-Tool	14.51±0.1	17.34	45.80±0.1	17.59
+ EnvCraft-Tool-Claw	24.19±0.2	11.26	**55.63**±0.3	13.25
+ EnvCraft-Claw	**24.85**±0.1	8.84	55.42±0.2	12.44
Qwen3-32B	22.92±0.4	17.34	60.35±0.3	23.16
+ EnvCraft-Claw	**29.51**±0.3	14.60	**62.78**±0.2	20.33
Qwen3.5-9B	47.34±0.1	11.50	72.22±0.1	19.49
+ EnvCraft-Claw	**52.19**±0.3	10.58	**76.30**±0.1	14.32

Table 3: Main results. Score denotes the average task score (%) and Token the average token usage per trajectory (K); $\pm\cdot$ is the standard deviation over three runs. Bold marks the best score per backbone group, and EnvCraft-Tool-Claw is the two-stage curriculum of Sec. 5.

work did not evaluate. We therefore treat these comparisons as reference points rather than as a controlled head-to-head evaluation, and revisit the implications of this below for τ^2-bench.

Evaluation. We evaluate on two categories of benchmarks. (1) two Claw-style agent benchmarks: PinchBench (Kilo AI Team, 2026) and ClawEval (Ye et al., 2026), which assess end-to-end capabilities in solving real-world, long-horizon tasks. We employ the standard OpenClaw harness for these evaluations. (2) two widely used multi-turn tool-use benchmarks, BFCL-v3 (Patil et al., 2025) and τ^2-bench (Yao et al., 2024), which evaluate core tool-use abilities essential for Claw Agents. All benchmarks require the model to interact with users and invoke tools through its native function-calling interface. We report the average score over three independent runs.

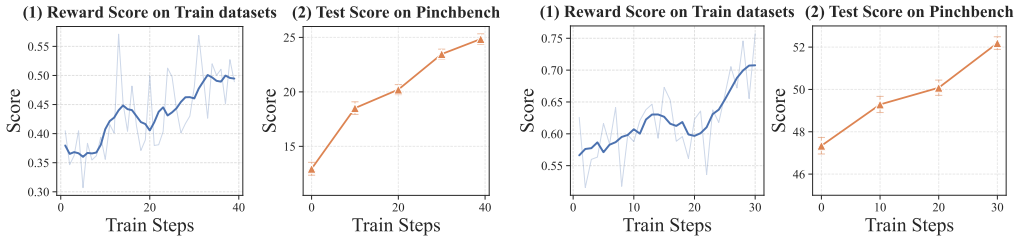
5.2 Main Results

Main Results on Claw-style Benchmarks. Table 3 reports performance on PinchBench and Claw-Eval, together with the average token cost per task. Training on EnvCraft-generated data consistently improves Qwen models across all scales and families. Concretely, for Qwen3-8B, Claw-specific training boosts PinchBench from 12.94% to 24.85% (+11.91 points) and Claw-Eval from 44.06% to 55.42% (+11.36 points). Qwen3-32B achieves gains of 6.59 points on PinchBench and 2.43 points on Claw-Eval. Qwen3.5-9B shows consistent gains of 4.85 points on PinchBench and 4.08 points on Claw-Eval. These improvements across three backbone configurations, ranging from 8B to 32B parameters, indicate that EnvCraft's benefits are not tied to a specific scale or architecture, underscoring the value of synthesized environments as a scalable training resource.

Comparing training data compositions, general tool-use data yields only modest gains (+1.57 and +1.74 points on PinchBench and Claw-Eval, respectively), whereas Claw-specific trajectories contribute the majority of improvements (+11.91 and +11.36 points, respectively). Sequential Tool-to-Claw training achieves comparable results (+11.25 and +11.57 points, respectively). We caution that this comparison of data compositions is observational rather than a controlled ablation, so we read it as a description of the recipe we adopt rather than as evidence for a specific data-ordering mechanism. Token cost separates the two compositions more sharply: EnvCraft-Tool inflates Qwen3-8B from 13.69K to 17.34K tokens per task on PinchBench (+26.7%) for a gain of only 1.57 points, whereas EnvCraft-Claw lowers the cost to 8.84K while adding 11.91 points, and the same ordering holds on Claw-Eval. Lower token usage is therefore not a generic consequence of training on synthesized data, but tracks the Claw-specific trajectories in particular.

Model	BFCL-v3 Multi-Turn					τ^2-bench			
	Base	Miss-Func	Miss-Param	Long-Context	Overall	Retail	Airline	Telecom	Overall
Advanced Models									
Gemini-3-Pro-Pre.	64.50	60.00	54.50	64.00	60.75	75.90	80.50	91.00	82.47
Claude Sonnet 4.5	69.00	65.00	52.50	59.00	61.37	72.40	72.00	84.90	76.43
Qwen3-235B	60.00	35.00	34.00	54.00	45.75	71.90	58.60	47.30	59.27
Kimi-K2-Instruct	57.50	35.00	42.00	49.00	45.88	70.60	56.50	65.80	64.30
Main Results									
Qwen3-8B	50.00	44.50	34.00	28.50	39.25	33.33	26.00	21.93	27.09
ScaleEnv-8B	-	-	-	-	-	50.90	37.50	27.20	38.53
EnvScaler	55.50	36.00	35.00	41.00	41.88	53.62	36.00	-	-
EnvFactory	-	-	-	-	49.00	43.86	44.00	13.16	33.67
AWM	-	-	-	-	-	41.23	38.50	23.47	34.40
EnvCraft-Tool	58.50	50.50	39.50	36.50	46.25	36.84	28.00	24.56	29.80
EnvCraft-Claw	55.00	52.00	40.50	36.00	45.88	35.52	26.50	23.68	28.57
Qwen3-32B	57.00	50.00	38.00	37.00	45.50	56.14	48.00	26.32	43.49
ScaleEnv-32B	-	-	-	-	-	63.60	48.00	30.90	47.50
EnvCraft-Tool	67.00	53.00	41.50	42.50	51.00	56.63	52.00	28.07	45.57
Qwen3.5-9B	52.00	53.50	39.50	34.00	44.75	35.14	32.00	15.79	27.64
EnvCraft-Tool	65.50	59.00	45.00	41.50	52.75	40.35	42.00	26.50	36.28

Table 4: Performance comparison of advanced models on multi-turn tool-use benchmarks (BFCL-v3 Multi-Turn and τ^2-bench). All scores are reported as percentages. "-" indicates unreported results in the original paper.



(a) EnvCraft-Claw on Qwen3-8B

(b) EnvCraft-Claw on Qwen3.5-9B

Figure 4: Training dynamics on EnvCraft-Claw environments. Test PinchBench scores are reported throughout RL, showing consistent and stable improvements for both (a) Qwen3-8B and (b) Qwen3.5-9B as training progresses.

Beyond end to end evaluation of EnvCraft, the appendix reports three targeted controls: an ablation of the tool-chain sampling strategy (Sec. F.2), a human audit of verifier reliability (Sec. F.1), and a decontamination audit against BFCL-v3 (Sec. F.3). Attribution across the remaining synthesis components is left to future work.

Training dynamics. Figure 4 illustrates the training dynamics of Qwen3-8B and Qwen3.5-9B during reinforcement learning on EnvCraft-Claw environments, reporting checkpoint-level test performance on PinchBench. Both models exhibit steady improvements throughout training: Qwen3-8B rises from 12.9% to 24.85%, while Qwen3.5-9B climbs from 47.3% to 52.2%. These consistent gains, across different scales and backbones, indicate that the learning signal provided by EnvCraft remains stable over the course of training rather than being confined to isolated checkpoints.

Results on Tool-Use Benchmarks. Table 4 presents results on BFCL-v3 and τ^2-bench. EnvCraft training consistently improves scores across both benchmarks. On BFCL, Qwen3-8B climbs from 39.25% to 46.25% under EnvCraft-Tool (+7.00 points) and achieves 45.88% under EnvCraft-Claw, matching the performance of Kimi-K2-Instruct (45.88%) despite being substantially smaller. Qwen3-32B gains +5.50 points, reaching 51.00%, while Qwen3.5-9B shows the largest boost, rising from 44.75% to 52.75% (+8.00 points). On τ^2-bench, gains

are markedly smaller: EnvCraft-Tool improves Qwen3-8B from 27.09% to 29.80% (+2.71 points), while EnvCraft-Claw yields 28.57%. We note explicitly that this is the one benchmark on which prior synthesis methods compare favorably to ours: ScaleEnv reports 38.53 at 8B, and EnvFactory and AWM report 33.67 and 34.40 at 8B, all above our 29.80. Consequently, under the unmatched-budget caveat noted above, we do not claim an advantage here.

A plausible explanation is that τ^2-bench evaluates conversational dual-control domains, where the difficulty lies in policy-constrained dialogue rather than workspace state mutation, whereas our synthesized tasks are constructed around terminal state-changing actions by design. Notably, EnvCraft-Claw achieves results on BFCL comparable to EnvCraft-Tool (45.88% vs. 46.25% for Qwen3-8B), suggesting that Claw-specific training does not degrade general tool-use ability.

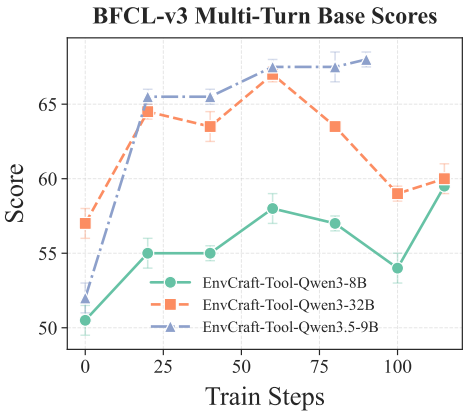


Figure 5: Training dynamics on the BFCL-v3 Base subset: Base score of Qwen3-8B, Qwen3-32B, and Qwen3.5-9B against RL training steps on EnvCraft.

Training dynamics. Figure 5 tracks the BFCL-v3 Base score of the three Qwen backbones as a function of RL training steps on EnvCraft. Because this figure monitors benchmark feedback across training, we state the checkpoint-selection rule explicitly: for every run we report the checkpoint with the highest training reward score, and benchmark feedback is not used for checkpoint selection. All three models improve over their step-0 initialization within the first ≈20 steps and stay above it for the remainder of training, reaching selected-checkpoint gains of 8.5, 10.0, and 13.5 points on Base (Table 4). Qwen3.5-9B improves most monotonically, whereas the two Qwen3 backbones exhibit mild mid-training oscillations but remain above their initial performance.

Inference Token-Cost. Figure 6 compares task performance and average token consumption on PinchBench before and after EnvCraft-Claw training. Averaged over the three backbones, EnvCraft-Claw reduces per-task token cost by 20% (14.18K → 11.34K) while simultaneously improving task scores. Specifically, Qwen3-8B drops from 13.69K to 8.84K tokens (−35%), Qwen3-32B from 17.34K to 14.60K (−16%), and Qwen3.5-9B from 11.50K to 10.58K (−8%). This score-up, cost-down pattern holds across all three backbones, and it is not the product of an explicit efficiency objective: the reward is purely outcome-based, with no length or process-level shaping term (Sec. D.3). The saving therefore reflects a change in how the policy reaches the goal state, rather than an incentive to emit fewer tokens.

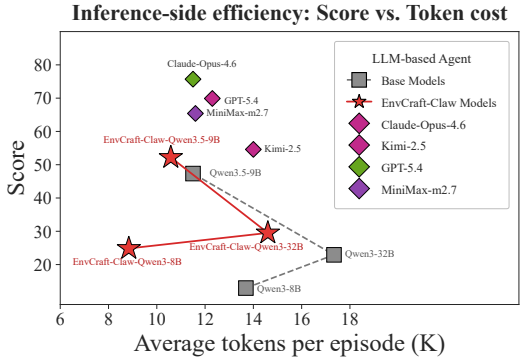


Figure 6: Inference efficiency: PinchBench score versus average token consumption per task before and after EnvCraft-Claw training.

6 Conclusion

We present EnvCraft, an end-to-end synthesis framework that automates the generation of executable environments and verifiable RL training tasks from natural-language descriptions, eliminating the need for manual environment engineering. We contribute 139 RL environments and $\sim 20K$ training instances at scale. Experiments demonstrate consistent gains on both Claw-style and general tool-use benchmarks, accompanied by reduced inference token costs. These results suggest that scalable, automated environment synthesis is a viable way to train claw-like agents.

AI USE STATEMENT

Generative AI models play two distinct roles in this work. First, they are part of the research method itself: the executable environments, training tasks, and verification scripts evaluated in this paper were synthesized by large language models of the DeepSeek series, and the resulting artifacts were filtered by the static and dynamic auditing protocols described in Secs. 3 and 4. Second, AI tools were used to refine the writing of this manuscript. In neither role were they used to formulate the scientific hypotheses, to interpret experimental results, to derive mathematical formulations, or to develop the scientific arguments presented in this work. The authors designed the synthesis and auditing pipeline and verified all reported results, and human annotators independently audited verifier reliability (Sec. F.1). The authors take full responsibility for the design of the study, the validity of the experiments, the interpretation of the results, and the accuracy and integrity of the manuscript.

REPRODUCIBILITY STATEMENT

The appendix specifies the full pipeline. For environment and task synthesis, we provide the scenario taxonomy and interaction archetypes (Sec. B.2), the generation prompts and schemas (Secs. C.1, C.2, C.3, C.5), and the execution and safety constraints (Sec. C.4). For training and evaluation, we provide the GRPO formulation (Sec. B.1), hyperparameters (Sec. D.1), the data curriculum and reward assignment (Secs. D.2, D.3), and the evaluation configuration (Sec. D.4). All synthesized environments are executable sandboxes with deterministic, programmatically verified rewards, so every reported result can in principle be reproduced by re-running the released pipeline. The synthesized environments, task data, and verification scripts will be released publicly upon acceptance.

LIMITATION

This paper has several limitations. First, although the synthesized environments are executable and undergo static and dynamic auditing, they remain simplified abstractions of real-world systems and may not fully capture external-service failures, nondeterministic behavior, evolving interfaces, or complex security constraints encountered in deployment. Second, the synthesis pipeline relies on LLM-generated specifications, dependency graphs, user intents, and verification scripts. Consequently, biases or errors introduced by the generator may propagate across different stages, while the predefined interaction archetypes and topology-based sampling strategy may underrepresent workflows outside the covered design patterns. Relatedly, our experiments evaluate the pipeline as a whole; apart from the sampling-strategy ablation (Sec. F.2), we do not isolate the marginal contribution of individual components such as archetype conditioning. Third, our state-based deterministic verifiers primarily evaluate whether the intended workspace mutations are completed, making them less suitable for tasks involving subjective quality, multiple acceptable outcomes, or purely read-oriented objectives. Finally, our training experiments focus on Qwen3 and Qwen3.5 backbones and a limited set of agent benchmarks. Evaluating broader model families, real production systems, and longer-running workflows remains an important direction for future work.

REFERENCES

Anthropic. Introducing claude opus 4.6. `https://www.anthropic.com/news/claude-opus-4-6`, February 2026.

Shihao Cai, Runnan Fang, Jialong Wu, Baixuan Li, Xinyu Wang, Yong Jiang, Liangcai Su, Liwen Zhang, Wenbiao Yin, Zhen Zhang, et al. Autoforge: Automated environment synthesis for agentic reinforcement learning. *arXiv preprint arXiv:2512.22857*, 2025.

Chen Chen, Xinlong Hao, Weiwen Liu, Xu Huang, Xingshan Zeng, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Yuefeng Huang, et al. Acebench: Who wins the match point in tool usage? *arXiv preprint arXiv:2501.12851*, 2025a.

Zhaorun Chen, Zhuokai Zhao, Kai Zhang, Bo Liu, Qi Qi, Yifan Wu, Tarun Kalluri, Sara Cao, Yuanhao Xiong, Haibo Tong, et al. Scaling agent learning via experience synthesis. *arXiv preprint arXiv:2511.03773*, 2025b.

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.

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

Tao Ge, Xin Chan, Xiaoyang Wang, Dian Yu, Haitao Mi, and Dong Yu. Scaling synthetic data creation with 1,000,000,000 personas. *arXiv preprint arXiv:2406.20094*, 2024.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Jujie He, Jiacai Liu, Chris Yuhao Liu, Rui Yan, Chaojie Wang, Peng Cheng, Xiaoyu Zhang, Fuxiang Zhang, Jiacheng Xu, Wei Shen, Siyuan Li, Liang Zeng, Tianwen Wei, Cheng Cheng, Bo An, Yang Liu, and Yahui Zhou. Skywork open reasoner 1 technical report. *arXiv preprint arXiv:2505.22312*, 2025.

Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan Arik, Dong Wang, Hamed Zamani, and Jiawei Han. Search-r1: Training llms to reason and leverage search engines with reinforcement learning. *arXiv preprint arXiv:2503.09516*, 2025.

Adam Tauman Kalai, Ofir Nachum, Santosh S Vempala, and Edwin Zhang. Why language models hallucinate. *arXiv preprint arXiv:2509.04664*, 2025.

Kilo AI Team. Pinchbench: Benchmarking system for evaluating LLM models as OpenClaw agents. `https://github.com/pinchbench/skill`, 2026. GitHub repository. Accessed: 2026-05-02.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328, 2022.

Kuan Li, Zhongwang Zhang, Huifeng Yin, Liwen Zhang, Litu Ou, Jialong Wu, Wenbiao Yin, Baixuan Li, Zhengwei Tao, Xinyu Wang, et al. Websailor: Navigating super-human reasoning for web agent. *arXiv preprint arXiv:2507.02592*, 2025a.

Xiaoxi Li, Jiajie Jin, Guanting Dong, Hongjin Qian, Yongkang Wu, Ji-Rong Wen, Yutao Zhu, and Zhicheng Dou. Webthinker: Empowering large reasoning models with deep research capability. *Advances in Neural Information Processing Systems*, 38:120091–120131, 2026a.

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

Yixia Li, Hongru Wang, Jiahao Qiu, Zhenfei Yin, Dongdong Zhang, Cheng Qian, Zeping Li, Xiaoteng Ma, Guanhua Chen, and Heng Ji. From word to world: Can large language models be implicit text-based world models? In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8084–8111, 2026b.

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*, 2025c.

MiniMax. Minimax m2.7: Early echoes of self-evolution. `https://www.minimax.io/news/minimax-m27-en`, March 2026.

Moonshot AI. Kimi k2.5: Visual agentic intelligence. `https://www.kimi.com/blog/kimi-k2-5`, January 2026.

Nous Research. Hermes agent: The agent that grows with you. `https://github.com/nousresearch/hermes-agent`, 2026. GitHub repository. Accessed: 2026-05-02.

OpenAI. Introducing gpt-5.4. https://openai.com/index/introducing-gpt-5-4/, March 2026.

OpenClaw Contributors. Openclaw: An open-source framework and repository. https://github.com/openclaw/openclaw, 2026. Accessed: 2026-06-29.

Shishir G. Patil, Huanzhi Mao, Charlie Cheng-Jie Ji, Fanjia Yan, 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 *Forty-second International Conference on Machine Learning*, 2025.

A. Prabhakar, Z. Liu, M. Zhu, J. Zhang, T. M. Awgalgaonkar, S. Wang, Z. Liu, H. Chen, T. Hoang, J. C. Niebles, S. Heinecke, W. Yao, H. Wang, S. Savarese, and C. Xiong. Apigen-mt: Agentic pipeline for multi-turn data generation via simulated agent-human interplay. In *Advances in Neural Information Processing Systems*, volume 38, 2025. doi: 10.52202/085713-2184.

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. *Advances in Neural Information Processing Systems*, 38:105523–105553, 2026.

Shuofei Qiao, Yanqiu Zhao, Zhisong Qiu, Xiaobin Wang, Jintian Zhang, Zhao Bin, Ningyu Zhang, Yong Jiang, Pengjun Xie, Fei Huang, et al. Scaling generalist data-analytic agents. *arXiv preprint arXiv:2509.25084*, 2025.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. Toolllm: Facilitating large language models to master 16000+ real-world apis. In *International Conference on Learning Representations*, volume 2024, pp. 9695–9717, 2024.

QwibitAI. Nanoclaw. https://github.com/qwibitai/nanoclaw, 2026.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Hesam Sheikh. Awesome openclaw usecases. https://github.com/hesamsheikh/awesome-openclaw-usecases, 2026. Accessed: 2026-05-03.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv: 2409.19256*, 2024.

Xiaoshuai Song, Haofei Chang, Guanting Dong, Yutao Zhu, Ji-Rong Wen, and Zhicheng Dou. Envscaler: Scaling tool-interactive environments for llm agent via programmatic synthesis. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 8326–8357, 2026.

Peter Steinberger. OpenClaw: Your own personal AI assistant. https://github.com/openclaw/openclaw, 2026. GitHub repository. Accessed: 2026-05-02.

M. Sullivan, M. Hartmann, and A. Koller. Procedural environment generation for tool-use agents. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 18544–18562. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.emnlp-main.936.

Tencent. Skillhub: Tencent cloud agent skill platform. https://skillhub.cloud.tencent.com/, 2026. Accessed: 2026-05-03.

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.

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

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.

Jialong Wu, Baixuan Li, Runnan Fang, Wenbiao Yin, Liwen Zhang, Zhenglin Wang, Zhengwei Tao, Ding-Chu Zhang, Zekun Xi, Robert Tang, et al. Webdancer: Towards autonomous information seeking agency. *Advances in Neural Information Processing Systems*, 38:120957–120985, 2026.

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*, 2026.

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.

Zhenghai Xue, Longtao Zheng, Qian Liu, Yingru Li, Xiaosen Zheng, Zejun Ma, and Bo An. Simpletir: End-to-end reinforcement learning for multi-turn tool-integrated reasoning. *arXiv preprint arXiv:2509.02479*, 2025.

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

Bowen Ye, Rang Li, Qibin Yang, Yuanxin Liu, Linli Yao, Hanglong Lv, Zhihui Xie, Chenxin An, Lei Li, Lingpeng Kong, et al. Claw-eval: Toward trustworthy evaluation of autonomous agents. *arXiv preprint arXiv:2604.06132*, 2026.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. Dapo: An open-source llm reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025.

Yirong Zeng, Xiao Ding, Yutai Hou, Yuxian Wang, Li Du, Juyi Dai, Qiuyang Ding, Duyu Tang, Dandan Tu, Weiwen Liu, et al. Tool zero: Training tool-augmented llms via pure rl from scratch. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 9135–9147, 2025.

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. *arXiv preprint arXiv:2505.00024*, 2025.

Ethical Statement

EnvCraft aims to reduce the cost of constructing executable environments and training data for agentic reinforcement learning. Nevertheless, the proposed capabilities involve potential risks. We therefore emphasize that EnvCraft is intended for controlled research environments. Generated environments and tasks should undergo security auditing, access control, and human review before release or deployment, particularly when they involve sensitive data, external services, or tools capable of producing consequential state changes.

A Related Works

Agentic Reinforcement Learning Recent research has increasingly shifted from supervised fine-tuning (SFT) toward Agentic RL for learning complex tool-use and multi-turn interaction policies. By enabling autonomous exploration in interactive environments and optimizing policies from rule-based or trajectory-level feedback, methods such as GRPO (Guo et al., 2025) and DAPO (Yu et al., 2025) improve model performance on challenging and out-of-distribution tasks. This closed loop between execution and learning allows agents to recover from errors and refine their behavior using environment feedback.

For general tool use, methods such as ToolRL (Qian et al., 2026), Tool-Zero (Zeng et al., 2025), and ToolN1 (Zhang et al., 2025) train LLMs to select and invoke tools by directly optimizing task

success. Search-oriented agents (Li et al., 2026a; Wu et al., 2026; Li et al., 2025a) interleave reasoning with real-time evidence retrieval to address challenging open-domain questions. In mathematical reasoning, ReTool, ToRL, and SimpleTIR (Feng et al., 2025; Li et al., 2025b; Xue et al., 2025) treat interpreters as executable action spaces. Similarly, CodeRL and DataMind (Le et al., 2022; Qiao et al., 2025) optimize code-generation policies using execution feedback. These developments further increase the demand for diverse, executable, and scalable training environments.

Executable Environment Synthesis Large-scale agent RL training demands low-latency, scalable, and reproducible environments. Environment construction can be broadly divided into direct production environments and synthesized environments. The latter can be further categorized into simulated environments and executable synthetic environments.

Production environments employ real-world APIs (Qin et al., 2024) and MCP servers (Xu et al., 2025) to provide authentic execution. However, they are expensive to scale and suffer from network latency and irreproducible states, which may destabilize RL training.

Simulated environments leverage LLMs to emulate tool behavior and state dynamics, enabling rapid prototyping (Chen et al., 2025b; Li et al., 2025c; 2026b). However, they are prone to hallucination and introduce additional inference cost and instability, limiting their applicability to large-scale training (Kalai et al., 2025; Wang et al., 2024).

Executable synthetic environments (Wang et al., 2026; Song et al., 2026; Xu et al., 2026; Tu et al., 2026; Fang et al., 2026) reconstruct tools and stateful backends through sandbox code generation, striking a balance among scalability, reproducibility, and execution fidelity.

Closely related lines of work generate tools and multi-turn tasks procedurally or through simulated interaction, e.g., procedural environment generation for tool-use agents (Sullivan et al., 2025) and agentic pipelines that synthesize verified multi-turn task blueprints (Prabhakar et al., 2025). These approaches target isolated tool endpoints and task-level blueprints, whereas our work focuses on *stateful workspace* environments that expose file systems, shell execution, and databases, and on composing them through a bi-level tool/environment dependency graph so that individual tasks require cross-environment execution.

B Additional Method Details

B.1 Group Relative Policy Optimization

We briefly review GRPO (Shao et al., 2024) in the language-generation setting. Formally, for each input instruction x, the policy π_θ samples a group of G candidate responses $\{y_i\}_{i=1}^{G}$. The optimization objective is defined as:

$$\bar{\rho}_{i,t} = \mathrm{clip}\left(\rho_{i,t}, 1-\epsilon_{\mathrm{clip}}, 1+\epsilon_{\mathrm{clip}}\right). \tag{1}$$

$$\mathcal{J}(\theta) = \mathbb{E}\left[\frac{1}{G}\sum_{i=1}^{G}\frac{1}{|y_i|}\sum_{t=1}^{|y_i|}\Big(\min\{\rho_{i,t}\hat{A}_i, \bar{\rho}_{i,t}\hat{A}_i\} - \beta D_{\mathrm{KL},i,t}\Big)\right]. \tag{2}$$

Here, y_i denotes the i-th sampled agent trajectory, and r_i is the outcome reward returned by the deterministic task verifier. Because the reward is assigned at the trajectory level, we use $\hat{A}_{i,t} = \hat{A}_i$ for all tokens in trajectory i.

B.2 Environment Scenario Taxonomy

We organize the collected environment domains into 83 fine-grained scenarios, including 41 general tool-use scenarios and 42 Claw-specific scenarios. Each fine-grained scenario may correspond to one or more executable environment instances. The "# Envs" column reports the number of synthesized environments associated with each scenario. Tables 5 and 6 provide the complete taxonomy and scenario descriptions.

No.	Fine-Grained Scenario	# Envs	Scenario Description
1	Messaging & Communication	5	APIs for instant messaging, mobile messaging, email, SMS, and other communication services, supporting session management, message sending and receiving, and user-status tracking.
2	Document & Note Management	3	APIs for document-management and personal note-taking systems, supporting document creation and editing, version control, note organization, and information retrieval.
3	Development Tools	1	Jupyter Notebook server APIs supporting the creation, execution, editing, and lifecycle management of computational notebooks.
4	Scheduling & Reminders	2	APIs for scheduling and reminder systems, supporting calendar-event management, timed reminders, recurrence configuration, and notification delivery.
5	Monitoring & Logging	3	APIs for network monitoring, model monitoring, and system-log management, supporting device-status detection, log collection, alerting, and performance-metric tracking.
6	Server & Application Management	1	Node.js application-server management APIs supporting application deployment, start and stop operations, process management, and runtime status monitoring.
7	Data Management	3	APIs for database management, data backup and recovery, and file-system operations, supporting persistent storage, data restoration, and structured file manipulation.
8	Memory & Storage	2	Key–value storage and memory-summarization APIs providing persistent information storage, retrieval, summarization, and archival-memory management.
9	Air Travel	1	Airline-ticketing APIs supporting flight search, seat reservation, payment processing, itinerary retrieval, and booking modification.

Table 5: Fine-grained general tool-use scenarios and their corresponding environment descriptions.

B.3 Interaction Archetype Specifications

To systematically address the complex operational demands inherent in OpenClaw scenarios, such as multi-tool orchestration, asynchronous agentic execution, and dynamic context shifts, we define a taxonomy of nine complementary interaction archetypes. Derived from core software interaction patterns in real-world OpenClaw deployments, these prototypes encapsulate the exhaustive spectrum of control flows and state evolution logic required for agent evaluation.

1. **Transactional Workspace Operations (CRUD & Auth):** Models authenticated entity workspaces—ticketing systems, content platforms, multi-tenant SaaS—where a session-bound user handle gates operations on a typed entity queue, each entity carrying an immutable identifier, mutable payload, and discrete status marker that progresses unidirectionally (open $\rightarrow$ in-progress $\rightarrow$ resolved $\rightarrow$ closed) with guard checks against illegal reversions and duplicate terminal closure; an append-only audit log records every mutation. Tools expose entity CRUD with schema validation, selective field updates, terminal-state resolution with mandatory metadata, session lifecycle primitives, and user-scoped filtered listing. The reference blueprint is a stateful workspace class: a module-level `DEFAULT_STATE` dict holds the entity list, identity counter, current-user variable, and log; lifecycle methods validate authentication and schema constraints before mutating; a `_load_scenario` entry point deep-copies defaults and applies overrides for harness snapshot-consistency.
2. **Cross-Tool Data Pipelines:** Simulates multi-stage ETL workflows—media monitoring, competitive intelligence, multi-source research synthesis—where heterogeneous raw inputs (text queries, audio, video, structured tables) are ingested through format-specific parsers into a unified intermediate representation, each job tracked with an auto-derived type tag (search, transcribe, parse) and processing status; completed jobs feed an aggregation stage that merges results with semantic deduplication across configurable keys, and a final rendering stage formats the consolidated set into markdown, bullet summaries, JSON, or graph descriptions. Tools provide source registration, multi-format ingestion, type-dispatched processing, cross-job aggregation with dedup-key configuration, and multi-

No.	Fine-Grained Scenario	# Envs	Scenario Description
10	Ground Transport	2	Car-rental and ride-hailing platform APIs supporting vehicle leasing, ride requests, driver and vehicle matching, route planning, and fare calculation.
11	Comprehensive Travel Booking	1	Integrated travel-booking APIs combining flight reservations, credit-card management, payment processing, and travel-budget constraints.
12	Accommodation	1	Hotel-booking APIs supporting hotel and room search, availability management, price calculation, booking creation, modification, and cancellation.
13	Vehicle Management	3	APIs for vehicle maintenance, motor-vehicle registration, and vehicle control, covering maintenance scheduling, registration records, and in-vehicle system operations.
14	Banking & Trading	3	APIs for retail banking, stock trading, and quantitative-trading bots, supporting account management, transfers, market-data retrieval, order placement, and automated trading.
15	Corporate Finance	1	Corporate financial-reporting APIs supporting financial-statement generation, multi-source data aggregation, accounting-period management, and compliance reporting.
16	Insurance	2	APIs for insurance-policy management and automobile-insurance claims, supporting policy lifecycle management, claim submission, evidence review, and settlement processing.
17	Reimbursement	1	Reimbursement and expense-claim management APIs supporting application submission, approval workflows, expense verification, and financial write-off processing.

Table 5: Fine-grained general tool-use scenarios and their corresponding environment descriptions (continued).

format output generation. The reference architecture is a four-phase pipeline engine (ingest $\rightarrow$ process $\rightarrow$ aggregate $\rightarrow$ render) where each phase produces an intermediate artifact; processing uses a dispatch table keyed by job-type to select among deterministic simulator functions; aggregation operates on a user-declared deduplication key with configurable string-normalization; rendering follows a format-strategy pattern.

3. **Dynamic Human-Agent Interaction:** Replicates multi-turn service conversations—customer-support triage, consultative sales, medical intake—where the agent must discover a hidden user persona (latent intent, floating-point confidence level, emotional valence and trend, confirmed and outstanding item sets) through progressive clarification, with a patience threshold forcing convergence or escalation when exceeded. State is organized into three layers: session records with turn counts, per-session message histories, and context-variable dictionaries; control flow runs a keyword-driven analysis pipeline that adjusts intent confidence (raised by confirmation language, lowered by clarification), transfers items from outstanding to confirmed on resolution detection, tracks mood deterioration beyond the patience bound, and generates deterministic mood-conditioned replies from a template bank seeded by message content and turn index. Tools expose session start with persona configuration, message send with auto-reply and context-delta reporting, history retrieval, manual context overrides, and session summarization with trajectory analysis. The blueprint is a three-layer state container with a pure analysis function mapping message text to context deltas and a deterministic template-based reply generator enabling reproducible evaluation.
4. **Asynchronous Event Triggering:** Captures event-condition-action automation—IFTTT/Zapier-style hubs, monitoring pipelines, smart-environment rule engines—where agents register heterogeneous event sources from a fixed taxonomy of external system categories, define rules binding a trigger type, optional source filter, and a condition predicate drawn from an operator set (equality, comparison, containment, regex matching) to typed action lists. Event injection is the sole triggering primitive: the event is validated, appended to an event log, then synchronously matched against all enabled rules with aligned trigger type and source filter; for each match, the condition is evaluated against the event payload and, when satisfied, all actions dispatch in definition order with results written to an action

No.	Fine-Grained Scenario	# Envs	Scenario Description
18	Online Shopping & Cart	2	APIs for online-shopping and retail platforms, supporting product browsing, cart management, promotional discounts, checkout, and order creation.
19	Product Catalog Management	1	E-commerce product-catalog APIs supporting product-information maintenance, category organization, attribute management, pricing, and inventory display.
20	Inventory & Order Management	2	APIs for inventory and order management, supporting inventory updates, order-status transitions, fulfillment tracking, and stock reconciliation.
21	Marketing & Reviews	2	APIs for product promotion and review systems, supporting campaign configuration, promotional-rule management, user reviews, ratings, and feedback moderation.
22	Task & Issue Tracking	3	APIs for task tracking, to-do management, and software-testing workflows, supporting assignment, prioritization, status transitions, and defect tracking.
23	Workflow & Job Scheduling	2	APIs for workflow-management and job-scheduling systems, supporting process definition, dependency configuration, job execution, retry handling, and status monitoring.
24	Content Management	1	Web content-management system APIs supporting page creation, editing, version control, approval, publication, and content lifecycle management.
25	Media Catalog & Playback	2	APIs for media-catalog databases and music-playback systems, supporting media indexing, metadata management, streaming playback, and playlist organization.

Table 5: Fine-grained general tool-use scenarios and their corresponding environment descriptions (continued).

log. Tools cover source registration, rule CRUD with full condition/action specification, event injection, and separate filtered queries over event and action histories for end-to-end traceability. The reference architecture is a synchronous ECA engine: rule evaluation is a linear scan performed immediately on injection; condition evaluation is a pure function with well-defined fallback semantics (string ops degrade gracefully, numeric ops fail closed); action execution is a log-only side effect; the dual-log design ensures full auditability.

5. **Constraint-Guided State Machine:** Represents systems governed by an explicitly declared finite state machine $\mathcal{S} \times \mathcal{A} \to \mathcal{S}'$—order lifecycles, loan origination, regulatory approval chains, device controllers—where machine definitions enumerate named states with on-enter/on-exit hooks, designated initial and terminal sets, and a transition table whose edges carry source, target, trigger-mode label, and optional nested guard predicates of the form `{field: {op: value}}` evaluated with multi-field AND semantics against per-entity data payloads. Entities are created at the initial state and transitioned by locating matching edges to the target, evaluating guards (fail-closed on missing fields), and on success updating current state, appending to visitation history, and logging; terminal-state entry marks the entity completed and blocks further transitions. Tools expose declarative machine definition (create, add states, add guarded transitions), entity lifecycle with three-result transition feedback (transitioned, blocked by guard, no such transition) and available-target enumeration, and filtered entity/transition inspection. The blueprint is a two-tier declarative engine: the machine tier is a pure graph definition with labeled edges; the entity tier is a runtime interpreter that walks the graph per entity; guard validation is a standalone pure function with explicit AND composition; trigger mode is metadata for agent interpretation.
6. **Multi-Resource Temporal Scheduling:** Simulates competitive time-axis resource allocation—calendar management, meeting-room booking, shift rostering—where agents manage multiple calendars each with timezone and working-hour constraints, and schedule events with ISO-8601 time bounds and integer priority into a global event list. Every addition triggers eager pairwise interval-overlap detection against non-cancelled events on the same calendar, marking conflicts automatically; resolution follows a four-strategy taxonomy: reschedule (assign new time bounds), cancel (terminal status), override (cancel all conflicting

No.	Fine-Grained Scenario	# Envs	Scenario Description
26	Social Media	1	Social-media publishing APIs supporting post creation, comment interaction, reposting, content retrieval, and follower-relationship management.
27	HR & Employee Management	1	Human-resources information-system APIs supporting employee-profile management, organizational structures, employment records, and payroll information maintenance.
28	Account & Membership	2	APIs for membership management and user registration, supporting account creation, authentication, membership-tier management, and access-control configuration.
29	Benefits & Wellness	1	Fitness-class booking APIs supporting class scheduling, member registration, coach assignment, capacity management, and booking cancellation.
30	Communication & Services	2	APIs for telecommunications and food-delivery services, supporting mobile-plan management, data-usage control, restaurant ordering, and delivery-status tracking.
31	Search & Information	1	Web-search APIs supporting keyword queries, result retrieval, result organization, search-history management, and bookmark collection.
32	Computation & Math	1	Mathematical-computation APIs supporting logarithmic, trigonometric, statistical, and numerical functions, together with configurable precision control.
33	Online Learning Platforms	1	Online-education platform APIs supporting course management, student enrollment, progress tracking, assignment handling, and Q&A interactions.

Table 5: Fine-grained general tool-use scenarios and their corresponding environment descriptions (continued).

events of strictly lower priority then promote), or split (truncate at a split point and spawn a continuation inheriting all metadata). Availability queries scan a target date within working hours, compute gaps via a cursor-based algorithm over sorted non-cancelled events, and return contiguous free intervals meeting a minimum duration. Tools cover calendar creation, event scheduling with auto-detection, explicit conflict inspection, four-strategy resolution, filtered event listing by calendar/status/date prefix, and availability discovery. The reference architecture is a temporal-constraint engine: conflict detection is a pure interval-overlap predicate; the resolution taxonomy is a strategy pattern with cascading override; timestamps use ISO-8601 string comparison for ordering with `datetime` parsing only for duration arithmetic.

7. **Multi-Criteria Bilateral Matching:** Evaluates two-sided market decision-making—resume-to-job matching, supplier sourcing, roommate pairing, marketplace recommendation—where agents configure matching pools with weighted criteria vectors (attribute-name to float-weight mappings) and a top-K cap, then populate supply and demand candidates with typed attribute dictionaries. Pairwise compatibility scoring normalizes per-attribute distances by type—numerical proximity via absolute distance, string equality with a partial-match fallback constant, list overlap as Jaccard ratio—and computes a weighted sum normalized to 0–100; results are sorted descending, truncated to top-K, and materialized as persistent match records that enter a two-phase lifecycle (pending $\rightarrow$ accepted or rejected) with explicit agent-mediated acceptance/rejection on behalf of the counterparty. Tools expose pool creation with weighted criteria, candidate addition, pairwise match execution with ranked scored output, re-ranking by score/supply/demand, accept/reject primitives, and filtered listing. The blueprint is a configurable scoring engine: the criteria vector is user-declared; scoring is a type-dispatched per-dimension calculator composed into a weighted-sum aggregator; matches are persistent records with an explicit status state machine; pools isolate independent matching contexts.
8. **Distributed Workflow Orchestration:** Replicates complex task-DAG execution—CI/CD pipelines, order fulfillment, content moderation, multi-service saga orchestration—where pipelines decompose work into ordered stage arrays, each stage specifying an action label, a

No.	Fine-Grained Scenario	# Envs	Scenario Description
34	Academic Administration	3	APIs for student-information and university course-selection systems, supporting student-profile management, course registration, enrollment constraints, and grade-record maintenance.
35	Knowledge Resources	1	Library-management APIs supporting catalog search, book borrowing, reservations, renewals, return processing, and overdue-fine calculation.
36	CRM & Customer Management	1	Customer-relationship management APIs supporting customer profiles, sales-opportunity tracking, interaction histories, and communication record maintenance.
37	Reservation & Booking	1	Restaurant-reservation APIs supporting restaurant and table information management, availability queries, booking creation, and dining-schedule modification.
38	Support Ticketing	1	Customer-support ticketing APIs supporting ticket creation, assignment, status updates, priority management, escalation, and service-queue management.
39	Health Records	1	Personal-health-record APIs supporting health-metric recording, vital-sign tracking, historical-data retrieval, and medical-record management.
40	Nutrition & Fitness	1	Nutrition-tracking and fitness-management APIs supporting dietary records, nutritional analysis, exercise logging, and workout-plan management.
41	Sensor & Device Management	1	IoT sensor-data platform APIs supporting device registration, sensor-data collection, device-status monitoring, and time-series data management.

Table 5: Fine-grained general tool-use scenarios and their corresponding environment descriptions (continued).

dependency list of prerequisite stage identities, an assigned worker (with role and capability tags), and a failure-policy tag (abort, skip, retry) with a retry cap. Execution is mode-dependent: sequential mode iterates stages in order, executing each pending stage whose dependencies are satisfied and halting on abort-policy failures; parallel mode identifies all dependency-satisfied pending stages and executes them concurrently. On failure the policy strategy applies: abort marks the pipeline failed; skip marks the stage skipped and continues; retry resets to pending with counter increment for explicit re-invocation. Rollback walks backward through completed stages invoking compensating actions; in-band worker messaging enables coordination. Tools cover worker registration/deregistration with busy-state protection, pipeline definition with sequential/parallel topology, stage-to-worker assignment, single-stage and full-pipeline execution, retry, rollback with compensation, result collection, and inter-worker messaging. The blueprint is a two-layer orchestration engine: the pipeline layer defines DAG topology through dependency lists; the execution layer enforces the dependency contract, manages worker state transitions (idle $\leftrightarrow$ busy), and delegates failure response to a policy-strategy pattern; rollback is a reverse-topological walk.

9. **Fault-Tolerant Stream Processing:** Simulates real-time asynchronous publish-subscribe message buses—telemetry ingestion, audit-log processing, IoT sensor hubs, event-sourcing backends—where typed event streams are declared with explicit schema contracts (field-name-to-type dictionaries), and subscribers attach through filtered subscriptions specifying a delivery mode (push, poll, batch) and an exact-equality filter condition over event payload fields. Publication pushes a typed payload with an assigned priority tier into the stream's append-only history, then fans out to all active matching subscriptions, placing event identifiers into per-subscription pending queues; explicit acknowledgment removes events from the pending queue, with unacknowledged events persisting indefinitely to simulate at-least-once redelivery semantics and create observable backpressure through queue depth. Unlike the event-driven pattern (synchronous single-fire rule evaluation), stream processing is inherently asynchronous and continuous: events flow indefinitely, subscriptions have lifecycles (create, filter, unsubscribe), and delivery guarantees require active consumer participation.

No.	Fine-Grained Scenario	# Envs	Scenario Description
1	Collaborative Office & Email Automation	3	Collaborative-office and email-automation environments supporting workflow orchestration for email-content extraction, automated social-media publishing, and event-driven email rules.
2	E-commerce & Logistics Management	5	E-commerce logistics environments supporting shipment tracking, return and exchange processing, inventory reconciliation, multi-stage logistics orchestration, and state-machine-based order management.
3	Spreadsheet Data Cleaning & Statistical Analysis	1	Spreadsheet-analysis environments supporting data cleaning, formula calculation, chart generation, pivot-table construction, and statistical analysis across workspace files.
4	Email Client Intelligent Management	1	Intelligent email-client environments supporting email sending and receiving, message organization, reply suggestion, and automatic extraction of actionable to-do items.
5	Sensitive Information Encryption & Management	1	Secure information-storage environments supporting encrypted password storage, credential retrieval, automatic form filling, and strong password generation.
6	Cross-Modal Content Transcription & Summarization	2	Cross-modal content-processing environments supporting audio- and video-to-text transcription, discussion consolidation, and multi-turn conversational summarization.
7	Personal Assistant Configuration	2	Personal-assistant environments supporting assistant-persona configuration, preference management, schedule coordination, and multi-turn dialogue orchestration.
8	Contact Information Intelligent Management & Maintenance	1	Intelligent contact-management environments supporting contact storage, tag-based classification, relationship maintenance, and context-aware reminders.
9	Travel Compliance & Booking Closure	5	Travel-compliance environments supporting policy verification, multi-platform price comparison, approval workflows, booking completion, and compliance-oriented state transitions.

Table 6: Fine-grained Claw-specific scenarios and their corresponding environment descriptions.

Tools expose typed stream creation with schema contracts, subscription management with filter and delivery-mode selection, prioritized event publication, acknowledgment, pending-event inspection, and event-history queries. The reference architecture is an asynchronous pub-sub bus: streams are schema-declared channels; subscriptions are independent consumer handles each with a private pending queue; the filter-matching function is a pure exact-equality predicate; acknowledgment is the sole offset-advancement mechanism.

C System Prompts and Execution Schemas

This section provides the system prompts and structured schemas used by EnvCraft for environment and task synthesis. To reduce inconsistencies between natural-language requirements and generated code, both synthesis stages follow explicit input–output contracts. The environment generator produces a standalone, session-isolated Python package, while the task generator constructs a self-contained workspace, a natural user request, and a deterministic verifier from a shared latent ground truth.

C.1 Environment Generation Prompt

Given an environment name, a workflow description, a set of domains, and structured world data, the environment generator is instructed to produce a complete Python package under `claw_envs/<environment_name>/`. The package must implement persistent session management, domain-specific tools, deterministic state transitions, executable evaluation logic, documentation, and concurrency tests.

The fixed system prompt used for environment generation is shown below. Fields enclosed by angle brackets are instantiated from the structured input specification described in Section C.2.

No.	Fine-Grained Scenario	# Envs	Scenario Description
10	Itinerary Disruption Dynamic Optimization	4	Dynamic itinerary-optimization environments supporting delay monitoring, automatic rebooking, linked hotel and airport-transfer adjustments, and traveler notification.
11	Travel Budget Auto-Accounting & Control	1	Automated travel-budget environments supporting expense categorization, budget allocation, threshold alerts, and spending-trend analysis.
12	Automated Reimbursement Reconciliation	5	Automated reimbursement environments supporting receipt recognition, bank-statement matching, expense-report completion, approval processing, and multidimensional reconciliation.
13	Travel Itinerary Offline Planning	1	Offline travel-planning environments supporting destination organization, itinerary construction, local information management, and multi-objective route optimization.
14	Multi-Dimensional Environmental Adaptive Regulation	3	Adaptive environmental-control environments supporting smart-device management, HVAC regulation, air-quality adjustment, and energy-consumption optimization.
15	Home/Office Security Patrol	4	Security-patrol environments supporting intrusion detection, scheduled inspection, alarm triggering, coordinated incident response, and evidence collection.
16	Environmental Data Real-Time Monitoring & Statistics	1	Real-time environmental-monitoring environments supporting sensor-data collection, threshold alerts, temporal trend analysis, and statistical report generation.
17	Smart Device Scheduled Task Management	1	Smart-device scheduling environments supporting access control, area monitoring, scheduled patrol tasks, rule execution, and emergency notification.
18	Investment Research Cross-Validation	4	Investment-research environments supporting financial-data comparison, stock-price verification, research-report generation, and real-time market-stream processing.

Table 6: Fine-grained Claw-specific scenarios and their corresponding environment descriptions (continued).

System Prompt for Environment Generation

```
You are an environment generator for reinforcement learning
training.  Synthesize a standalone Python package under
claw_envs/<environment_name>/ that satisfies the following contract.
Required artifacts.
1.  Environment class.  Generate a Python class that owns the
session lifecycle, orchestrates domain modules, records actions with
deterministic timestamps, and exposes evaluation entry points.  It
must implement:
       create_session(scenario_id) -> session_id
       reset_session(session_id) -> None
       get_session_state(session_id) -> dict
       evaluate_session(session_id) -> dict
The returned session state must be compatible with the verifier.
The evaluation method must return per-dimension scores and an
overall_score.
2.  Tool functions.  Generate one Python module for each domain,
such as mail.py, social.py, or tickets.py.  Each module exposes
functions that take a session dictionary as input, return structured
dictionaries, and mutate the session state in place.  Domain
functions must not parse command-line arguments, acquire file locks,
or access the file system directly.
3.  State initialization.  Provide either a module-level
DEFAULT_STATE dictionary or an initial_session() factory.  Every
newly created session must be initialized from this template.  The
template must include:
```

No.	Fine-Grained Scenario	# Envs	Scenario Description
19	Industry Sentiment & Growth Analysis	1	Industry-analysis environments supporting competitor monitoring, strategy tracking, market-report generation, growth analysis, and user-feedback aggregation.
20	System Patch Injection Defense	1	System-security environments supporting sensitive-key isolation, prompt-injection detection, suspicious patch inspection, and security-audit alerting.
21	Identity Forgery & Audit Fraud	1	Identity-verification and audit-defense environments supporting auditor-identity validation, anomalous-behavior detection, forgery-risk assessment, and early warning.
22	Bulk Unauthorized Data Export Interception	1	Data-loss-prevention environments supporting sensitive-data identification, export-approval workflows, permission verification, and unauthorized-transfer blocking.
23	Internal Core Document Leak Interception	1	Document-security environments supporting confidentiality classification, outbound-transfer detection, access-policy validation, and real-time leak prevention.
24	End-to-End Interview Process Hosting	1	Recruitment-workflow environments supporting resume screening, interview scheduling, candidate communication, evaluator coordination, and end-to-end status tracking.
25	Employee Onboarding Automated Pipeline	1	Employee-onboarding environments supporting account creation, asset allocation, permission configuration, document processing, and automatic training-task triggering.
26	Employee Performance Auto-Assessment Archiving	1	Performance-management environments supporting multi-source employee data aggregation, score calculation, report generation, and persistent assessment archiving.

Table 6: Fine-grained Claw-specific scenarios and their corresponding environment descriptions (continued).

```
        session_id; scenario_id; created_at;
        meta.base_time; meta.action_index = 0;
        workspace_account; one state container per domain;
        actions = [].
4. Documentation. Generate both README.md and SKILL.md. The README
must describe the task domain, session model, trainer bootstrap
procedure, agent-visible commands, and stress-testing procedure.
The SKILL file must provide a concise English workflow guide, list
the preferred tools and commands, and state critical behavioral
constraints. It must contain fewer than 50 lines.
5. Persistence logic. Generate a store.py module implementing:
        load_session(state_root, session_id) -> dict
        save_session(state_root, session_id, session_dict) ->
        None
        create_session(state_root, session_id, initial_state) ->
        None
        acquire_lock(state_root, session_id) -> Lock
        release_lock(lock) -> None
Session state must be stored at <state_root>/<session_id>/session.json,
with the corresponding lock at <state_root>/<session_id>/.lock. All
writes must use an atomic temporary-file-and-replace procedure, and
all reads must remain isolated across sessions.
6. Exception handling. Every public-facing environment method,
CLI command, tool function, and storage operation must return a
structured error dictionary rather than propagating an exception.
Infrastructure failures must be mapped to safe task-side messages.
Returned errors must not expose raw tracebacks, internal file-system
paths, connection strings, or implementation details. Domain-level
validation errors must contain sufficient information for the agent
to correct its action.
```

No.	Fine-Grained Scenario	# Envs	Scenario Description
27	Employee Offboarding Automated Process	1	Employee-offboarding environments supporting permission revocation, asset-return confirmation, workspace-data backup, account deactivation, and completion tracking.
28	Automated Domain Technical Survey	1	Automated technical-survey environments supporting paper retrieval, source organization, domain-trend analysis, evidence synthesis, and survey-report generation.
29	Literature Citation Relationship Archiving	1	Literature-management environments supporting citation-graph construction, citation-impact analysis, related-work discovery, and persistent relationship archiving.
30	Multi-Group Experiment Metrics Auto-Comparison	1	Experiment-analysis environments supporting experimental-configuration recording, metric aggregation, group comparison, statistical difference analysis, and result archiving.
31	Open-Source Reproduction Record Archiving	1	Reproduction-management environments supporting procedural record keeping, dependency and environment configuration, result validation, and reproducibility archiving.
32	Music & Audio Playback	1	In-vehicle audio environments supporting music search, playback control, queue manipulation, playlist management, and media-status retrieval.
33	Vehicle Control & Status Management	1	Vehicle-control environments supporting air-conditioning, seat, window, lighting, driving-mode, and multimedia operations together with vehicle-status inspection.
34	Navigation & Trip Planning	1	In-vehicle navigation environments supporting destination search, route planning, real-time traffic retrieval, charging-station selection, and trip adjustment.

Table 6: Fine-grained Claw-specific scenarios and their corresponding environment descriptions (continued).

7. Test cases. Generate concurrency_test.py to validate session isolation, concurrent-write serialization, successful evaluation of a correct workflow, penalties for stale or forbidden actions, and correct separation between hidden trainer commands and agent-visible commands.
8. Verifier-compatible state interface. The environment class must expose:

```
get_env_state() -> dict
_load_scenario(scenario: dict, long_context: bool =
False) -> None
```

The former returns the complete internal state as a plain dictionary. The latter deep-copies DEFAULT_STATE and applies scenario overrides so that the harness can deterministically initialize, snapshot, and restore the environment.
Design constraints.
One rollout corresponds to one session identifier, and the trainer owns the complete session lifecycle. The agent must never receive or manually provide a session identifier. Trainer bindings are injected through the environment variables <ENV_NAME>_SESSION_ID, <ENV_NAME>_STATE_ROOT, and <ENV_NAME>_SCENARIO_ID.
Immutable world data must be stored under data/ and must not be modified during a rollout. Mutable state must be represented as file-backed JSON with atomic writes and file locks. Timestamps must be derived from the scenario base time and action index rather than from the wall clock.
All agent-visible commands must return structured JSON. Hidden trainer commands, including prepare-rollout and reset-rollout, must not appear in agent-facing help text.

No.	Fine-Grained Scenario	# Envs	Scenario Description
35	Infrastructure Automated Operations	1	Infrastructure-operations environments supporting server-fault detection, diagnostic inspection, automatic recovery, service verification, and operations-record maintenance.
36	Cloud Resource Cost Auto-Audit	1	Cloud-cost auditing environments supporting multi-project cost aggregation, abnormal-spending detection, budget alerts, and cost-optimization recommendations.
37	Online Fault Root-Cause Postmortem Archiving	1	Incident-management environments supporting fault-timeline construction, root-cause analysis, corrective-action tracking, and postmortem knowledge-base archiving.
38	Brand & Product Research	1	Brand- and product-research environments supporting SKU-information retrieval, competitor-feature comparison, evidence aggregation, and market-positioning analysis.
39	Multimedia Clue Tracking	1	Multimedia-investigation environments supporting cross-document clue association, evidence-chain construction, provenance tracking, and information tracing.
40	Churn Warning & Proactive Retention	1	Customer-retention environments supporting churn-risk assessment, retention-strategy generation, customer segmentation, and automated customer communication.
41	Customer Tier Label Auto-Classification	1	Customer-classification environments supporting behavior analysis, tier-rule configuration, eligibility evaluation, and automatic customer-label updates.
42	Scheduled Business Report Auto-Generation	1	Scheduled reporting environments supporting Markdown-template completion, multi-source data aggregation, visualization generation, and scheduled report publication.

Table 6: Fine-grained Claw-specific scenarios and their corresponding environment descriptions (continued).

C.2 Environment Input and Output Schemas

The environment generator receives a structured specification rather than an unconstrained natural-language request. The input contains the target domains, immutable world data, scenario definitions, evaluation rules, and optional execution constraints.

```
{
  "environment_name": "string",
  "task_description": "string",
  "domains": ["string"],
  "world_data": {
    "accounts": {
      "<account_id>": {
        "name": "string",
        "role": "string",
        "persona": "string"
      }
    },
    "contacts": {
      "<contact_id>": {
        "name": "string",
        "org": "string",
        "role": "string"
      }
    },
    "scenarios": [
      {
        "scenario_id": "string",
        "task_prompt": "string",
        "base_time": "ISO-8601 string",
        "gold_facts": ["string"],
        "stale_facts": ["string"],
```

```
        "required_attachments": ["string"],
        "required_replies": ["string"],
        "allowed_claims": ["string"],
        "forbidden_claims": ["string"],
        "scoring_rules": {
          "<dimension_name>": {
            "weight": "float",
            "rule": "string"
          }
        }
      }
    ],
    "attachments": {
      "<attachment_id>": {
        "filename": "string",
        "content": "string"
      }
    },
    "domain_artifacts": {
      "<domain>": [
        {
          "id": "string",
          "fields": {}
        }
      ]
    }
  },
  "constraints": {
    "max_turns": "optional integer",
    "required_reading_order": ["string"],
    "forbidden_action_sequences": [
      ["string"]
    ]
  }
}
```

The `gold_facts` field defines information that must be used in a successful trajectory, whereas `stale_facts` contains outdated or distracting information that must be rejected. Similarly, `allowed_claims` and `forbidden_claims` provide deterministic constraints for the evaluator. The `scoring_rules` field defines the evaluation dimensions and their weights.

The generator returns the complete package as a structured collection of files, together with explicit state, command, evaluator, and test descriptions.

```
{
  "environment_name": "string",
  "documentation": "README.md content",
  "skill_md": "SKILL.md content",
  "files": [
    {
      "path": "relative package path",
      "content": "complete file content",
      "role": "environment_class | domain_module | cli |
               evaluator | repository | store |
               concurrency_test | doc | data_scenario |
               data_accounts | data_contacts |
               data_attachment | data_artifact"
    }
  ],
  "session_schema": {
    "session_id": "string",
    "scenario_id": "string",
    "created_at": "ISO-8601 string",
```

```
    "meta": {
      "base_time": "string",
      "action_index": 0
    },
    "workspace_account": {},
    "<domain>": {},
    "actions": [
      {
        "action_index": 0,
        "timestamp": "string",
        "action_type": "string",
        "details": {}
      }
    ]
  },
  "cli_commands": {
    "hidden": [
      {
        "name": "string",
        "description": "string",
        "bindings_output": ["string"]
      }
    ],
    "agent_visible": [
      {
        "name": "string",
        "description": "string",
        "output_format": {
          "status": "string",
          "data": {}
        }
      }
    ]
  },
  "evaluator": {
    "dimensions": [
      {
        "name": "string",
        "weight": "float",
        "description": "string"
      }
    ],
    "aggregation": "weighted_sum",
    "deterministic": true
  },
  "concurrency_test": {
    "scenarios": ["string"],
    "assertions": ["string"]
  }
}
```

The explicit `role` attached to every generated file enables automatic completeness checking. An environment is rejected if any required artifact, interface, or test category is absent.

C.3 Unified Tool Schema

Each generated tool is represented using a unified declarative schema. Besides conventional names, descriptions, and parameter definitions, the schema explicitly records whether a tool mutates state and which state fields may be affected. These annotations are subsequently used to distinguish read and write tools and to analyze tool dependencies.

```
{
  "name": "string",
```

```
  "description": "string",
  "parameters": {
    "type": "object",
    "properties": {
      "<param_name>": {
        "type": "string | integer | number |
                boolean | array | object",
        "description": "string",
        "default": "optional value"
      }
    },
    "required": ["string"]
  },
  "returns": {
    "type": "object",
    "description": "string",
    "success_keys": ["string"],
    "error_format": {
      "error": "human-readable, agent-safe message"
    }
  },
  "side_effect": "read | write",
  "modifies_state": "boolean",
  "state_fields_affected": ["string"]
}
```

A parameter without a `default` field is treated as required when it is listed in `parameters.required`. The `side_effect`, `modifies_state`, and `state_fields_affected` fields provide an explicit operational description beyond the natural-language tool documentation.

C.4 Execution and Safety Constraints

The generated package must satisfy the following executable constraints.

Errors are divided into infrastructure failures and domain-level validation failures. Infrastructure errors are converted into generic messages that do not expose internal paths or implementation details. Domain errors identify the invalid entity, missing field, current state, or valid alternative whenever this information is necessary for agent self-correction.

Representative error behavior is summarized in Table 8.

C.5 Task Generation Prompt

For Claw-specific file-workspace tasks, the task generator receives a sampled operational skeleton consisting of a tool chain, environment domain, gold actions, and data schema. It then generates four mutually consistent artifacts in one pass:

1. a task configuration file;
2. a natural user request;
3. an initial-workspace construction script; and
4. a deterministic verification script.

The task prompt, workspace state, and verifier must all be derived from a single internally established ground truth. This one-shot constraint prevents mismatches in artifact paths, target records, numeric results, and evaluation conditions.

#	Constraint	Applicable Components
1	All public functions return dictionaries and map failures to structured error responses rather than propagating exceptions.	Environment, tools, CLI, storage, evaluator
2	No mutable module-level global state is permitted; all mutable data must reside in the session or environment state.	All modules
3	The `data/` directory is immutable during rollout and can be read only through the repository layer.	Repository and domain modules
4	Session updates use an atomic temporary write followed by `os.replace`.	Storage layer
5	File locking uses platform-supported locking with a bounded timeout and an agent-safe error response.	Storage layer
6	Timestamps are computed from `base_time` and `action_index`; wall-clock timestamps are forbidden.	Environment and domain modules
7	CLI outputs follow a structured success or error schema.	CLI
8	Agent-visible commands obtain session bindings only from trainer-injected environment variables and do not expose a `--session-id` argument.	CLI
9	Hidden trainer commands are excluded from agent-facing help messages.	CLI
10	Domain modules operate only on the provided session dictionary and do not perform CLI parsing, file access, or lock acquisition.	Domain modules
11	The evaluator is a deterministic, side-effect-free function with no randomness or external I/O.	Evaluator
12	The environment exposes complete state retrieval and deterministic scenario-loading interfaces for harness snapshot and restoration.	Environment class

Table 7: Code-level constraints enforced during environment generation.

The core system prompt is given below.

System Prompt for File-Workspace Task Generation

```
You are an AI-agent evaluation architect, scenario writer, and
full-stack engineer.  Given a business skeleton consisting of a tool
chain and an environment domain, generate a highly discriminative and
purely code-verifiable file-based workplace task for task <task_id>.
Before producing any output, internally establish one ground truth.
Generate the following four files in one pass:

       tasks/<task_id>.yaml
       tasks/prompts/<task_id>.md
       tasks/<task_id>/env_builder.py
       scripts/<task_id>/verify_workplace.py

All four files must describe the same objective and unique answer.
The business objective in the user prompt must be derivable from
the workspace generated by env_builder.py.  Every artifact, field,
path, and value checked by verify_workplace.py must correspond to
an explicit requirement in the user prompt.  The verifier must not
introduce a second answer or check an artifact that the prompt does
not request.
Output the four files in the required order.  Wrap each file in a
Markdown code block whose first line is its complete relative path.
```

Category	Example	Required Behavior
Missing binding	Session environment variable is absent	Return an error indicating that the trainer must initialize the rollout.
Session not found	The requested session does not exist	Return an agent-safe message stating that the session may have expired or been reset.
Lock timeout	Another worker holds the session lock	Request a retry without exposing the lock path.
Permission failure	The current account cannot modify an entity	Identify the affected entity and state that write permission is absent.
Invalid transition	An entity is already in the requested terminal state	Report the current state and explain why the transition is invalid.
Missing field	A required command argument is absent	Name the missing field and describe the corresponding requirement.
Invalid reference	A thread, document, or entity identifier does not exist	Identify the invalid reference and request correction.
Stale information	The agent relies on an outdated artifact	Allow the tool to emit a warning and let the deterministic evaluator apply the corresponding penalty.
Infrastructure failure	The state root cannot be accessed	Return a generic system error while retaining full diagnostic information only in server-side logs.

Table 8: Structured error categories used in generated environments.

Do not emit explanations, headings, separators, or scores outside the four code blocks.
Task configuration. The YAML file must follow the OpenClaw task schema and include the prompt path, the asset identifier, and basic runtime configuration.
User prompt. Write the request as a realistic email, ticket, or verbal handoff from a specific workplace role. The prompt should communicate the background, problem, and business objective in the character's voice.
Do not expose hidden answers, expected values, variable names, verifier logic, scoring rules, or a numbered solution procedure. When referring to files, use workspace-relative paths only. The request should describe the desired business outcome while requiring the agent to infer the necessary operations.
Workspace construction. Use only the Python standard library. Construct a professional and non-trivial initial workspace containing distractors, dirty data, near-duplicate records, outdated versions, decoy entries, and domain-specific non-standard file formats.
The workspace must nevertheless imply exactly one objectively correct answer. Distractors may increase reasoning difficulty but must not create ambiguity. The current working directory is already the task asset directory, so all paths must be relative. File paths created by the builder must match those referenced by the prompt and verifier exactly.
Deterministic verification. The verification script must use only the Python standard library and must not invoke an LLM, issue network requests, or read mock API credentials. It should evaluate objective properties such as:

- exact numeric computation;
- record extraction, filtering, deduplication, and merging;
- classification and threshold decisions;
- set equality and ordering;
- file or directory structure;
- schema and format validity.

```
Open-ended writing quality must not be used as a primary evaluation
target.  When textual output is required, score only structurally
extractable objective elements.
Structured files must be parsed using native JSON, CSV, path, numeric,
or regular-expression operations.  Do not use fuzzy substring
matching when exact parsing is available.  Unexpected fields or
fabricated entities should receive substantial penalties.
Use a fine-grained weighted score rather than a binary outcome.  The
script must write the final result to workplace_score.json using the
following structure:

      {"total_score":  85,
      "details":  [
      {"item":  "...", "score":  10, "max_score":  10,
      "passed":  true, "reason":  "..."}
      ]}

The verifier receives the workspace path as its first command-line
argument, defaulting to the current directory.
The supplied tool chain, gold actions, environment domain, and data
schema serve only as the underlying business skeleton.  Do not
mention internal tool names, CLI commands, or implementation-specific
execution commands in the user-facing task.
```

C.6 Cross-Artifact Consistency

The task generator is constrained to produce all task components in a single response because independent generation can introduce inconsistencies between the natural-language request, the initial workspace, and the verifier. EnvCraft therefore enforces the following cross-artifact invariants:

1. **Path consistency.** Every required output path in the user request must exactly match the path inspected by the verifier.
2. **Data consistency.** Every expected value checked by the verifier must be derivable from the files generated by the environment builder.
3. **Objective consistency.** The verifier may assess only outputs and conditions explicitly implied by the user request.
4. **Answer uniqueness.** The initial workspace must admit one objectively correct result, despite containing distractors and noisy records.
5. **Information separation.** Hidden answers, scoring criteria, and verification logic must remain absent from the user-facing prompt.
6. **Code-only evaluation.** The final reward is computed by deterministic programmatic checks rather than by an LLM-based judge.

Together, the explicit environment contract, unified tool schema, and cross-artifact task-generation protocol make generated samples directly executable and deterministically verifiable. They also enable automated auditing of missing package components, invalid tool interfaces, non-isolated state mutations, inconsistent artifact paths, and non-deterministic reward logic.

D Experimental Results

Hyperparameter	Value
Optimization	
Optimizer	AdamW
Learning rate	1×10^{-6}
Learning-rate scheduler	Constant
Warmup ratio	0.0
Weight decay	0.01
Gradient clipping	1.0
Training epochs	1
Global batch size	64
GRPO	
Group size G	8
Clipping coefficient ϵ_{clip}	0.2
KL coefficient β	0.02
Advantage normalization	Group-wise normalization
Entropy coefficient	0.0
Rollout	
Rollout batch size	64
Rollouts per prompt	8
Temperature	1.0
Top-p	1.0
Maximum prompt length	4,096
Maximum generation length	32,768
Maximum action turns	32
Infrastructure	
Number of GPUs	64 NPUs (8 nodes $\times$ 8 NPUs)
GPU type	Huawei Ascend 910B NPU
Tensor parallel size	4
Data parallel size	8 rollout replicas
Gradient checkpointing	Enabled

Table 9: Common hyperparameters used for EnvCraft reinforcement learning. Model-specific differences are described in the main text.

D.1 Training Hyperparameters

We train all models using Group Relative Policy Optimization (GRPO) implemented with the VERL framework (Sheng et al., 2024). Unless otherwise specified, all experiments use the same optimization, rollout, and evaluation configurations. The detailed hyperparameters are summarized in Table 9.

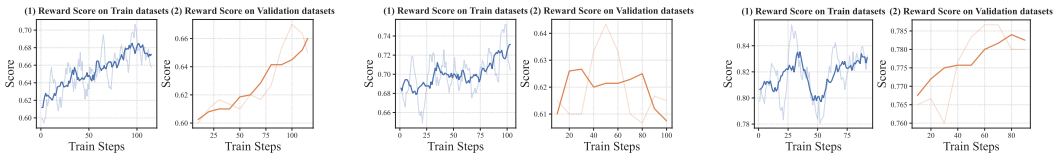
D.2 Training Data and Curriculum

We consider two training strategies. In direct Claw-specific training, the model is optimized solely on the EnvCraft-Claw dataset. In the two-stage Tool-to-Claw curriculum, the model is first trained on EnvCraft-Tool and subsequently optimized on EnvCraft-Claw.

For the first stage, we train the model for `115` steps using `8k` EnvCraft-Tool samples. The resulting checkpoint is then used to initialize the second stage, which is trained for `30` steps using `2k` EnvCraft-Claw samples. The learning rate is 1×10^{-6} in the first stage and 1×10^{-6} in the second stage. All other hyperparameters are shared across the two stages.

D.3 Reward Assignment and Rollout Handling

Each rollout receives an outcome reward from the programmatic verifier associated with its training sample. For EnvCraft-Tool, we adopt a binary reward: a successfully completed trajectory receives a reward of 1.0, whereas an unsuccessful trajectory receives a reward of 0.0. For EnvCraft-Claw, the verifier assigns a normalized reward in the range $[0, 1]$ according to the degree of task completion,



(a) Qwen3-8B (b) Qwen3-32B (c) Qwen3.5-9B

Figure 7: Training dynamics of Qwen3-8B, Qwen3-32B, and Qwen3.5-9B on EnvCraft-Tool. The left panels show training performance, while the right panels report performance on 50 held-out tasks excluded from training.

where 1.0 indicates full completion and 0.0 indicates complete failure. Intermediate values represent partial task completion. Unless otherwise specified, no intermediate process-level reward is assigned.

A rollout is considered unsuccessful if the agent produces an invalid tool call, exceeds the maximum number of action turns, reaches the execution timeout, triggers an unrecoverable environment exception, or fails to make verifiable progress toward the target state. Recoverable tool-execution errors remain part of the interaction trajectory and do not immediately terminate the rollout, allowing the policy to learn error-recovery behavior. The final reward is assigned after the resulting environment state is evaluated by the corresponding deterministic verifier. Verifier execution errors are treated as failed verification and the corresponding rollout is assigned a reward of 0.0.

D.4 Evaluation Configuration

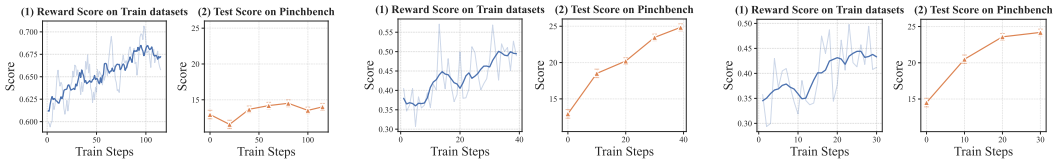
For evaluation, all models are decoded using a temperature of `1.0`, top-p of `0.6`, and a maximum generation length of `32768` tokens. Each task permits at most `32` agent turns and has an execution timeout of `300 seconds`. We use `8` independent runs per model and report the `mean` score. The reported error terms denote `standard deviation` across `evaluation runs`.

For PinchBench and Claw-Eval, task success is determined using the official evaluation protocol. For BFCL-v3 and τ^2-bench, we follow the benchmark-provided evaluation scripts and default task splits. All models use the same decoding and execution settings unless the official model interface imposes different constraints.

D.5 Training Dynamics and Data Composition

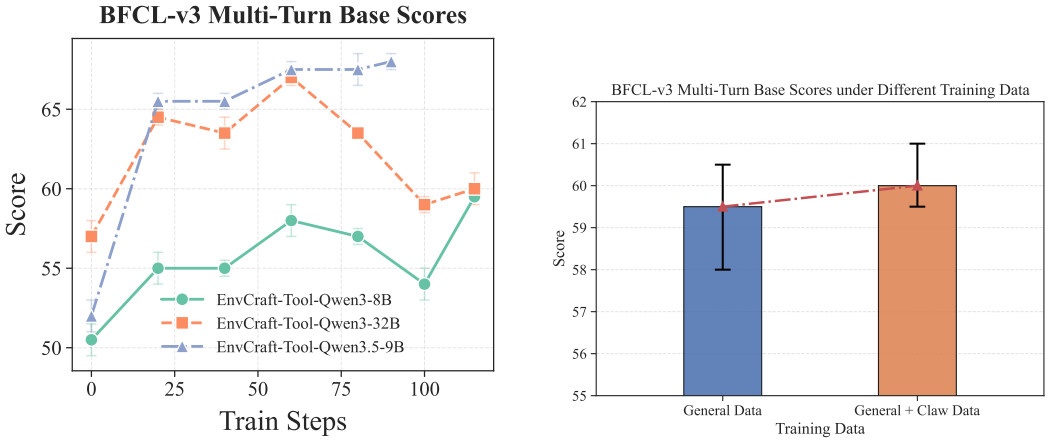
Training Dynamics on EnvCraft-Tool. We examine the training dynamics of Qwen3-8B, Qwen3-32B, and Qwen3.5-9B on EnvCraft-Tool. As shown in Figure 7, the left panels report training performance, while the right panels evaluate intermediate checkpoints on a held-out set of 50 EnvCraft-Tool tasks reserved before training and excluded from optimization. All three backbones exhibit overall improvements on both splits, despite checkpoint-level fluctuations, indicating that EnvCraft-Tool provides consistent learning signals rather than merely encouraging memorization of training instances.

Effect of Training Data Composition. We investigate how different data compositions affect claw-task performance on Qwen3-8B (Fig. 8). General tool-use training yields modest gains: PinchBench rises +1.57 and Claw-Eval rises +1.74 points. In contrast, Claw-specific training delivers substantially larger improvements, reaching +11.91 and +11.36 points, respectively. Sequential Tool-to-Claw training achieves comparable results (+11.25 and +11.57 points, respectively). These results indicate that domain-aligned Claw trajectories, which provide stateful workspaces, long-horizon execution, and cross-tool dependencies, contribute the majority of system-level gains. General tool-use data offers complementary supervision for basic function selection and parameter grounding, yet its standalone effect is limited. Overall, the findings suggest that task-aligned executable trajectories are more critical than simply accumulating heterogeneous training data.



(a) General Tool-use training (b) Claw-specific training (c) Tool-to-Claw training

Figure 8: Effect of training-data composition on Qwen3-8B. We compare general tool-use training, Claw-specific training, and sequential general-to-Claw training using checkpoint-level performance on PinchBench.



(a) General tool-use training (b) Continual Claw-specific training

Figure 9: Retention of general tool-use capability during Claw-specific reinforcement learning. Scores are measured on the BFCL-v3 Multi-Turn Base subset at different training checkpoints.

Retention of General Tool-Use Capability. A potential concern is that specializing a model on Claw-specific environments may degrade its general tool-use capability. To examine this issue, we evaluate intermediate and final checkpoints on the BFCL-v3 Multi-Turn Base subset. Figure 9 compares general tool-use training with subsequent Claw-specific training. After switching to Claw-specific environments, the model maintains a BFCL score of 60.0, compared with 59.5 before the transition, while its PinchBench performance improves from 14.51 to 24.19. These results provide evidence that Claw-specific reinforcement learning improves system-level execution without measurable degradation on the BFCL-v3 Multi-Turn Base subset.

D.6 Data Quality Analysis

LLM-Based Quality-Auditing Rubric. We employ an LLM-based inspector to assess the logical validity, executability, and internal consistency of each synthesized task before difficulty calibration. For each candidate, the inspector receives four inputs: the complete environment source code, the initialized environment state, the user intent sequence, and the corresponding programmatic validation protocol. The inspector then evaluates the candidate along three dimensions: task solvability and logic, validation-protocol integrity, and cross-component data consistency.

Specifically, we use **DeepSeek-V4-Pro** as the auditing model. Each candidate is inspected once using deterministic decoding, and the three dimension scores returned by the inspector are directly used for quality filtering. In addition to numerical scores, the inspector produces a concise, dimension-specific justification to facilitate failure analysis. Table 10 presents the complete scoring rubric.

Let $s_{\text{sol}}, s_{\text{val}}, s_{\text{con}} \in \{0, 1, 2, 3\}$ denote the scores for task solvability, validation integrity, and data consistency, respectively. A candidate is retained only if

Dimension	Score 3	Score 2	Score 1	Score 0
Task Solvability & Logic	The task is logically valid, non-trivial, and fully solvable under the initial state and environment rules. Completing it requires meaningful interaction, with no state conflicts.	The task is theoretically solvable but is either trivial, requires no meaningful interaction, or contains minor parameter mismatches that a robust agent may bypass.	The task is unsolvable because its requested actions or target state contradict the initial state, environment constants, or transition rules.	The task substantially hallucinates entities, tools, APIs, or states that are absent from the environment.
Validation-Protocol Integrity	The validation protocol is executable and completely checks the intended success conditions, without evident false-positive or false-negative paths.	The protocol is executable, but its assertions are incomplete or weak; for example, it verifies that an action occurred without checking the required final state.	The protocol contains syntax or runtime errors, references undefined variables or APIs, or is expected to fail during execution.	The validation protocol is missing or evaluates conditions unrelated to the user task.
Data Consistency & Syntactic Cleanliness	Entity names, identifiers, value types, and schemas are fully consistent across the environment state, user task, and validation protocol.	The components are semantically consistent but contain minor type or formatting mismatches that do not fundamentally alter the task objective.	Critical fields are missing, or explicit contradictions exist between the user-facing description and the underlying environment configuration.	The task record contains malformed JSON, severely corrupted text, or otherwise unusable structured data.

Table 10: Complete rubric used by the LLM-based data-quality inspector. Each synthesized task is assigned an integer score from 0 to 3 along the three dimensions.

$$\begin{aligned} s_{\text{sol}} &= 3, \\ s_{\text{val}} &= 3, \\ s_{\text{con}} &\geq 2, \\ s_{\text{sol}} + s_{\text{val}} + s_{\text{con}} &\geq 8. \end{aligned} \tag{3}$$

This strict gate requires every retained task to be fully solvable and equipped with a reliable programmatic reward function, while permitting only minor formatting inconsistencies that do not affect execution. Candidates failing any of these conditions are removed before the subsequent rollout-based difficulty calibration stage.

The auditing prompt provides the inspector with the raw environment code, initial state, user intent sequence, and validation script. It explicitly defines the score anchors in Table 10 and requires the inspector to return only a structured JSON object containing dimension-specific analyses, numerical scores, and the final qualification decision. A compact representation of the required output format is shown below.

Listing 1: Structured output schema of the LLM-based quality inspector.

```
{
  "reasoning_steps": {
    "dimension_1_analysis":
      "Evidence-based analysis of task solvability.",
    "dimension_2_analysis":
      "Evidence-based analysis of validation integrity.",
    "dimension_3_analysis":
      "Evidence-based analysis of data consistency."
  },
  "scores": {
    "task_solvability": 0,
    "validation_integrity": 0,
    "data_consistency": 0
  },
  "is_qualified": false
```

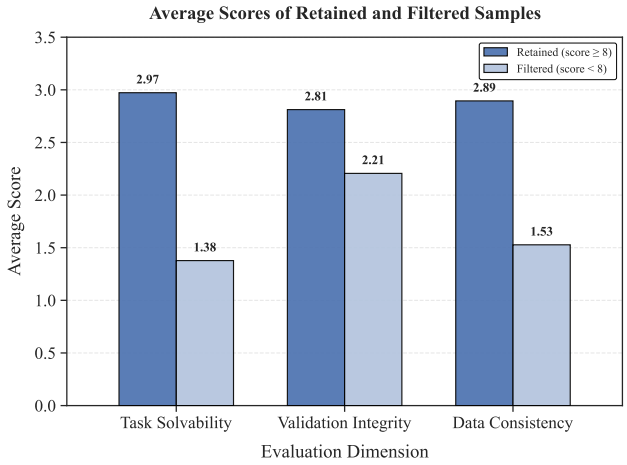


Figure 10: Average LLM-rubric scores of retained and filtered task candidates across task solvability, validation-protocol integrity, and data consistency. Each dimension is scored on a 0–3 scale. A candidate is retained only when it receives scores of 3, 3, and at least 2 on the three dimensions, respectively.

16 }

Effectiveness of LLM-Based Quality Auditing. We further examine whether the LLM-based inspector meaningfully distinguishes high-quality tasks from invalid candidates. Figure 10 compares retained and filtered samples in terms of their average task-solvability, validation-integrity, and data-consistency scores.

Retained samples achieve consistently higher scores across all three dimensions, with the largest separation observed in task solvability. In particular, the mean rubric score, averaged across the three dimensions, increases from 1.38 for filtered candidates to 2.97 for retained samples. Because retained candidates must satisfy the strict conditions in Equation 3, their scores are concentrated near the upper end of the rubric. These results indicate that the auditing procedure does not merely enforce syntactic correctness, but also filters tasks with contradictory execution logic, inconsistent environment states, or unreliable reward functions.

Difficulty Calibration. After rubric-based filtering, we calibrate task difficulty by evaluating each candidate task with Qwen3.5-27B over eight independent rollouts. All rollouts use the same system prompt, user instruction, tool definitions, and initial environment state, while using different random seeds to induce independent trajectories. We adopt a sampling temperature of 0.7 and a top-p value of 0.95 for all rollouts.

Let $z_j \in \{0, 1\}$ denote the outcome of the j-th rollout. A rollout is considered successful ($z_j = 1$) only if it terminates within the maximum interaction budget and the deterministic verifier confirms that the resulting environment state satisfies all task constraints. A rollout is considered unsuccessful ($z_j = 0$) if the verifier returns a negative result, the agent reaches the maximum of 32 action turns without completing the task, or the rollout exceeds a wall-clock timeout of 300 seconds. The empirical pass rate of a task is computed as

$$p = \frac{1}{8} \sum_{j=1}^{8} z_j. \tag{4}$$

We retain only tasks satisfying

$$0 < p < 1, \tag{5}$$

corresponding to tasks solved in one to seven of the eight rollouts. Tasks with $p = 1$ are removed because they provide limited learning signals, whereas tasks with $p = 0$ are excluded because they are likely unsolvable, underspecified, or excessively difficult for the current training stage.

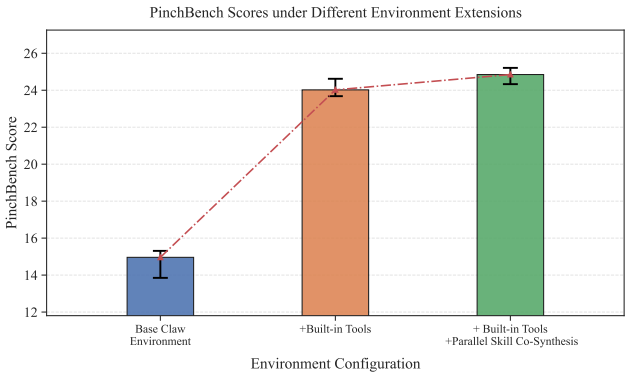


Figure 11: Ablation study of Claw-specific environment components on PinchBench. The three configurations progressively introduce built-in system tools and parallel skill co-synthesis. All models use the same Qwen3-8B backbone and training configuration.

We distinguish verifier rejection from verifier failure. A verifier that executes normally but returns a negative result is treated as a failed rollout. In contrast, verifier runtime exceptions, malformed outputs, or verifier timeouts are treated as data-quality defects rather than agent failures. Such samples are removed from difficulty calibration and routed to the verifier-repair and re-auditing pipeline. We allow at most two automatic repair attempts and discard samples whose verifiers remain invalid afterward.

D.7 Ablation Studies

Contribution of Claw-Specific Environment Extensions. To examine the contribution of the Claw-specific environment design, we train Qwen3-8B under three progressively enhanced configurations and evaluate the resulting models on PinchBench. As shown in Figure 11, augmenting the base Claw environment with built-in system-level tools substantially improves the score from 14.96 to 24.02, demonstrating that direct access to executable workspace operations provides the primary learning signal for system-level agent capabilities. Further introducing parallel skill co-synthesis increases the score to 24.33, yielding a smaller but consistent additional gain. These results suggest that built-in tools are the dominant contributor, while synthesized skills provide complementary workflow guidance that helps agents more effectively utilize the available execution interfaces. The best performance is achieved when both components are combined, validating the effectiveness of the complete Claw-specific environment design.

E Qualitative Trajectory Example

Cross-File Performance Aggregation. We present a representative EnvCraft-Claw trajectory involving multi-file workspace inspection, structured data integration, and programmatic state mutation. The agent must identify valid records across an employee roster, a monthly output ledger, and role-specific scoring rules, while excluding incomplete or inconsistent entries. It then computes weighted performance scores, writes the results to a designated workspace file, and verifies the generated artifact. This example illustrates how EnvCraft tasks require agents to combine information retrieval, constraint checking, numerical computation, and persistent file operations within a single executable workflow. For readability, we omit repetitive intermediate reasoning and abbreviate lengthy tool outputs while preserving the original action sequence and execution results.

F Additional Studies

F.1 Verifier Reliability Audit

Setup. We sample 100 tasks uniformly at random from the released task pool, stratified into 60 Claw-specific and 40 general tool-use tasks, and sweep at least one task from every interaction archetype. For each task, two annotators independently write a reference solution that should be

Representative EnvCraft-Claw Trajectory: Performance Score Aggregation

User Request. Linda, an HR specialist, asks the agent to prepare a monthly performance report. The workspace contains an employee roster under `data/employees/`, monthly output records under `data/ledgers/`, and role-specific scoring rules under `data/rules/`. Some records may be incomplete or redundant. The agent must retain only employees who: (1) appear in the roster, (2) have a monthly output record, and (3) have a role with a defined scoring rule. It must compute their weighted scores and save the resulting report to `performance/scores_202501.json`.

Step 1: Inspect the workspace

Agent rationale. First, identify the available data sources and confirm that the target output directory exists.

Tool calls:

```
ls(path="data")
ls(path="performance")
```

Observation. The workspace contains three data directories: `employees`, `ledgers`, and `rules`. The `performance` directory is currently empty.

Step 2: Load the relevant records

Agent rationale. Read the employee roster, monthly output ledger, and scoring-rule files to determine their schemas and join keys.

Tool calls:

```
read("data/employees/employees.json")
read("data/ledgers/monthly_outputs.json")
read("data/rules/scoring_rules.json")
```

Observation. The files use `employee_id` as the employee-level join key and `role_code` as the rule-level join key. Scoring weights are available for Software Engineers (`SE`) and Quality Assurance staff (`QA`), but not for Human Resources staff (`HR`).

Step 3: Resolve inconsistent records

Agent rationale. Cross-reference the three sources and retain only records satisfying all eligibility conditions.

- `emp001`, `emp002`, and `emp003` are valid across all three sources.
- `emp004` is excluded because the `HR` role has no scoring rule.
- `emp005` is excluded because no monthly output record is available.
- `emp999` is excluded because it does not appear in the employee roster.

For each eligible employee, the agent computes

$$s = w_{\text{delivery}} x_{\text{delivery}} + w_{\text{quality}} x_{\text{quality}} + w_{\text{collaboration}} x_{\text{collaboration}}.$$

Figure 12: A representative EnvCraft-Claw trajectory involving workspace inspection, cross-file record matching, rule-based computation, persistent state mutation, and output verification. For readability, repetitive intermediate reasoning and lengthy tool outputs are omitted while preserving the original action sequence and execution results.

Representative EnvCraft-Tool Trajectory: Performance Score Aggregation (Continued)

Step 4: Generate the performance report

Agent rationale. Use a Python script to join the records, calculate the weighted scores, sort employees by total score, and write a structured JSON artifact.

Tool call:

```
execute_bash(python scoring script)
```

Observation. Three eligible employees are identified:

Employee	Role	Weighted Score
Alice (`emp001`)	SE	92.0
Bob (`emp002`)	SE	84.5
Charlie (`emp003`)	QA	80.0

The results are written to `performance/scores_202501.json`.

Step 5: Verify the workspace mutation

Agent rationale. Read the generated file to verify that the requested artifact exists and contains the expected records and summary statistics.

Tool call:

```
read("performance/scores_202501.json")
```

Observation. The file contains three employee records, with two `SE` employees and one `QA` employee. The scores and ordering match the role-specific weighted calculations.

Final Response. The performance report has been generated successfully and saved to `performance/scores_202501.json`. Three employees satisfy all eligibility conditions: Alice with a score of 92.0, Bob with 84.5, and Charlie with 80.0. Diana is excluded because no HR scoring rule exists, Eve because her monthly ledger is missing, and `emp999` because the identifier is absent from the employee roster.

Figure 12: Representative EnvCraft-Claw trajectory continued.

Task family	Tasks	False accept	False reject	Verifier error
Claw-specific	60	3 (5.0%)	2 (3.3%)	2 (3.3%)
General tool-use	40	2 (5.0%)	1 (2.5%)	1 (2.5%)
Overall	100	5 (5.0%)	3 (3.0%)	3 (3.0%)

Table 11: Verifier reliability audit on 100 sampled tasks. Inter-annotator agreement: $\kappa = 0.87$.

Condition	PinchBench	Claw-Eval
Topology (default)	24.85	55.42
Uniform	19.40	51.10
Singleton	22.80	53.60

Table 12: Sampling-strategy ablation at a matched nominal task budget. All runs use Qwen3-8B and identical hyperparameters.

accepted, and construct up to three negative solutions that each differ from the reference in exactly one respect: an unfinished sub-goal, a correct result written to the wrong workspace path, and a plausible but incorrect value. Every solution is executed against the frozen verifier inside the sandbox, and the verifier's decision is compared with the annotator label. We report the false-accept rate (negative solution accepted), the false-reject rate (reference solution rejected), the verifier-error rate (exception, timeout, or malformed output), and inter-annotator agreement (Cohen's κ).

Results. Table 11 summarizes the audit. The overall false-accept rate of 5.0% is the most consequential figure for training: one in twenty negative solutions is rewarded, which places an upper bound of roughly 95% on the fidelity of the reward signal on this sample. False accepts occur at the same rate in both task families (5.0% each), so the error is not concentrated in the Claw-specific setting. False rejects occur at 3.0% and penalize solutions that reach the required terminal state through a different but legitimate sequence of operations. At 3.0%, verifier errors (exceptions, timeouts, or malformed output) are routed to the repair-and-re-audit path rather than scored as agent failures, so they reduce effective sample yield without corrupting the reward. Because false accepts outnumber false rejects, the verifier is mildly permissive rather than strict, which biases the learning signal toward accepting partially completed work; reducing the false-accept rate is therefore the priority for the next iteration of the audit.

F.2 Topology-Aware Sampling Ablation

Setup. We fix the environment set, the backbone (Qwen3-8B), the task budget, and all training hyperparameters, and vary only the strategy that samples tool chains from the bi-level tool/environment dependency graph. *Topology* (our default) uses weighted random walks over the graph. *Uniform* draws chains from the same length distribution but samples tools uniformly from the global pool, ignoring graph edges and weights. *Singleton* generates single-tool tasks with no cross-environment composition. We train one model per condition and report PinchBench and Claw-Eval.

Results. Table 12 reports the comparison. Topology-aware sampling outperforms uniform chain sampling by 5.45 points on PinchBench and 4.32 points on Claw-Eval at an identical nominal task budget. The ordering of the two control conditions is the more informative result: uniform chaining (19.40) falls below the singleton condition that uses no chaining at all (22.80), so adding multi-tool composition is not beneficial by itself. Chains whose arguments cannot be satisfied by the preceding state appear to inject noise that more than offsets the additional task complexity they introduce. The topology-aware condition still improves on the singleton condition by 2.05 points on PinchBench, indicating that valid cross-environment composition carries a genuine additional signal beyond tool-call complexity alone. We therefore read the benefit as arising from chain validity rather than from chain length. Because uniform sampling also produces fewer chains that survive validity filtering, part of the gap may reflect a smaller effective training set rather than the sampling strategy itself; we do not separate the two effects here.

F.3 Decontamination Audit

Setup. We audit the 41 general-domain scenarios against the BFCL-v3 evaluation set at three levels of granularity: tool-name overlap, domain-name overlap, and instance-level overlap, measured as the longest common substring between a generated task instruction and any BFCL-v3 evaluation instance. We then remove every environment flagged at any level and retrain Qwen3-8B on the remaining general subset under an identical budget. Per-scenario overlap details are released with the code rather than tabulated here.

Results. Of the 41 general scenarios, 12 share at least one tool name with BFCL-v3 and 7 share a similar domain name. At the instance level, 78 of 19,777 tasks (0.4%) exceed a 20-token longest common substring against a BFCL-v3 instance, with a maximum of 41 tokens. Removing all flagged environments discards 7 environments and 2,400 tasks. On BFCL-v3, Qwen3-8B scores 39.25 before training, 46.25 under the full general subset, and 43.81 under the decontaminated subset.

Analysis. Instance-level overlap is small in absolute terms: 0.4% of tasks exceed a 20-token longest common substring against a BFCL-v3 instance. The decontaminated model retains 4.56 of the 7.00-point BFCL gain, so overlap with the evaluation set does not fully explain the improvement. The loss of 2.44 points after decontamination is nevertheless larger than the measured instance-level overlap alone would suggest, for two reasons. First, the removal is coarse: discarding an entire environment whenever any of its tools or its domain name resembles a BFCL-v3 entry removes 2,400 tasks, 12% of the corpus, including many tasks with no instance-level overlap. Second, domain-name overlap (7 of 41 scenarios) is considerably more frequent than instance-level overlap and may reflect shared task structure rather than verbatim reuse. The observed drop therefore mixes genuine contamination with a training-set-size effect, and we do not treat 43.81 as a clean estimate of uncontaminated transfer. We report it as the conservative lower bound, and note that separating the two effects would require matching the decontaminated run on task count. The main text reports the full-subset setting; this audit provides the complementary conservative view.