From Agent Skills to Executable Environments: Frontier-Targeted Training for Tool-Use LLMs

Anonymous ACL submission

Abstract

LLMs increasingly use external tools and stateful environments for complex, multi-step tasks. Further improving their tool-use capabilities, however, remains bottlenecked by the lack of scalable stateful environments and verified multi-turn trajectories matched to the target LLM's current capability boundary. We observe that *agent skills*, packages pairing workflow specifications with executable scripts, provide scalable seed material for tool-use environment synthesis. This motivates **Grafting**, a skill-to-environment synthesis loop in which coding-agent harnesses perform open-ended construction under deterministic validation contracts, producing self-contained executable environments that are refined using trajectory-level verdicts. In an initial end-to-end run, Grafting produces 500 environments and approximately 12K verified multi-turn trajectories. We train on these trajectories with **Fathoming**, an iterative GRPO procedure that selects verified trajectories near the target model's capability boundary. Training a 4B-parameter instruction-tuned model with two rounds of Fathoming on the trajectory pool produces FATHOM-4B and yields strong out-of-distribution gains: BFCLv4 multi-turn accuracy rises from 22.0 to 39.7 (+17.7 points), and ComplexFuncBench per-call accuracy rises from 49.6 to 61.6 (+12.0 points). Using the same environment-synthesis pipeline, we further construct GROVE-5K, a collection of 5,101 executable environments. We will open-source the codebase and generated environments.

1 Introduction

Tool-augmented large language models (LLMs) are increasingly deployed as agents that interact with external tools in stateful environments (Qu et al., 2025; Schick et al., 2023; Qin et al., 2023). Solving such tasks requires sequential decision making over tool invocations and environment state across multiple interaction turns (Yao et al., 2024). A common route is to synthesize tool-interaction data and then train on it with supervised or reinforcement fine-tuning (Liu et al., 2024b,a; Prabhakar et al., 2025; Qian et al., 2025). For multi-turn tool use, however, the most useful data should capture grounded interaction dynamics: trajectories should be generated in scalable executable environments, verified against environment state, and aligned with the target LLM's current capability boundary. Obtaining such environments and trajectories with both quality and scale remains challenging.

Existing work constructs tool-use data and environments from several starting points. Many pipelines begin with tool, API, or domain specifications, then synthesize tool-calling examples or materialize executable backends (Liu et al., 2024b,a; Prabhakar et al., 2025; Tu et al., 2026; Song et al., 2026; Cai et al., 2025). Benchmark-oriented systems instead connect agents to real or simulated services (Xu et al., 2025; Guo et al., 2024; Li et al., 2025b). These directions have substantially improved scale and coverage, but it remains difficult to jointly obtain environments that are reproducible, resettable, and stateful enough for grounded multi-turn interaction, and to use the resulting trajectories in a way that matches the target model's current capability boundary.

To address the first challenge, we look for seed material that already contains operational structure rather than only tool names or domain labels. We observe that *agent skills* (Agent Skills Project, 2026) provide such an intermediate representation: a skill typically packages a workflow specification, executable scripts, and reference material. This makes skills **workflow-bearing** through multi-step procedures, **implementation-anchored** through bundled executable logic, and **domain-grounded** through tasks that users already want agents to automate. These properties make skills promising seeds for scalable stateful environments, but raw skills are not directly usable as training environments.

Turning skills into reproducible tool-use environments requires more than wrapping existing scripts. We propose **Grafting** (Section 3.1), a validation-driven skill-to-environment synthesis loop that starts from filtered skills with substantive workflow and implementation signals. First, Grafting treats each skill as a workflow-level seed rather than a runtime dependency, and coding-agent harnesses such as Claude Code (Anthropic, 2025) or Codex (OpenAI, 2025) synthesize self-contained environments with tool schemas, database state, backend code, documentation, and tests under deterministic validation gates. Second, it generates grounded tasks, simulates multi-turn trajectories through the deterministic backend, and verifies them by replaying tool calls against fresh environment state. Third, environment evolution uses trajectory-level verdicts to improve the environment through repair or enrichment. This validation-driven pipeline enables Grafting to scale diverse executable environments and grounded multi-turn trajectories while preserving reproducibility and resettable state.

To address the second challenge, we propose **Fathoming** (Section 3.2), an iterative training procedure that uses the verified trajectory pool according to the target LLM's current capability boundary. Fathoming slices each accepted trajectory at tool-call depths, probes the target model with multiple completions per slice, and uses pass and format rates to bucket examples by suitability for training. It then trains with Group Relative Policy Optimization (GRPO) (Shao et al., 2024) on frontier-near data and re-probes the updated model before the next selection round. This loop keeps training data near the model's capability boundary and provides informative signal for progressive improvement.

Our contributions are as follows:

- We propose **Grafting**, a skill-to-environment synthesis loop that uses coding-agent harnesses under deterministic validation contracts to build self-contained executable environments and refine them using trajectory-level verdicts.
- We validate the end-to-end Grafting-to-Fathoming pipeline on an initial 500-environment run, producing approximately 12K verified multi-turn trajectories. Training Qwen3-4B-Instruct-2507 (Qwen Team, 2025) with two rounds of Fathoming produces FATHOM-4B, improving BFCLv4 multi-turn accuracy from 22.0 to 39.7 (+17.7 points) and ComplexFuncBench per-call accuracy from 49.6 to 61.6 (+12.0 points).
- Using the same environment-synthesis pipeline, we construct and characterize GROVE-5K: 5,101 portable executable environments with broad domain and tool diversity, released with provenance metadata for tool-use data scaling.

2 Related Work

2.1 Environment Synthesis for Agent Training

Training tool-use agents requires executable environments and verifiable interaction trajectories. Recent work automates environment construction at scale, differing mainly in *what they start from* and *how they execute tools*.

Most pipelines bootstrap from lightweight seeds. EnvScaler (Song et al., 2026) and AutoForge (Cai et al., 2025) start from domain keywords or tool descriptions, ScaleEnv (Tu et al., 2026) from procedural tool-graphs, AgentScaler (Fang et al., 2025) from clustered real-world APIs, and AgentWorld (Dong et al., 2026) and TOUCAN (Xu et al., 2025) from live MCP servers; others need no seed, with Simia (Li et al., 2025b) and GenEnv (Guo et al., 2025) inventing tool behavior with an LLM. Coverage thus traces back to designer vocabulary, API catalogs, or unconstrained LLM imagination; we instead seed from developer-published agent skills, whose workflows and executable scripts provide richer operational grounding.

These pipelines also differ in execution: deterministic local backends (Cai et al., 2025; Song et al., 2026; Tu et al., 2026; Wang et al., 2026), LLM-simulated responses (Li et al., 2025b; Guo et al., 2024, 2025), or live API passthroughs (Xu et al., 2025). Simulation and passthrough sidestep construction cost but introduce nondeterminism or service drift and hold no persistent state. Grafting uses deterministic backends and adds a trajectory-driven evolution loop that feeds verdict statistics back to repair or enrich environments, a mechanism prior pipelines do not employ.

2.2 Tool-Use Data and Trajectory Synthesis

A parallel line synthesizes tool-use training data at scale, including verification-based pipelines such as APIGen (Liu et al., 2024b), APIGen-MT (Prabhakar et al., 2025), and ToolACE (Liu et al., 2024a), and graph-based methods such as Magnet (Yin et al., 2025) and ToolMind (Yang et al., 2025).

Unlike these approaches, our trajectories are simulated in self-verified executable environments with authoritative state, then replayed by a separate validation agent against a fresh database copy for grounded verification.

2.3 Reinforcement Learning for Tool Use

RL has become a powerful paradigm for tool use beyond supervised fine-tuning. ToolRL (Qian et al., 2025) gives the first systematic study of reward design within GRPO; Agent-R1 (Cheng et al., 2025) extends the MDP formulation to multi-turn tool-using agents; ReTool (Feng et al., 2025) uses outcome-based RL to learn *when* to invoke tools during reasoning; ToRL (Li et al., 2025a) and Tool Zero (Zeng et al., 2025b) show that tool use can emerge from pure RL on base models without SFT; and Tool-Star (Dong et al., 2025) adds hierarchical rewards across tool types. A common thread is that RL is applied to a *fixed* dataset.

A complementary line instead adapts training data or task difficulty to the learner's evolving capability. LoopTool (Zhang et al., 2025) is the closest parallel, using difficulty probing and closed-loop training. ToolACE-R (Zeng et al., 2025a) re-selects training data via a model-aware difficulty metric, GenEnv (Guo et al., 2025) co-evolves an environment simulator that calibrates task difficulty, Environment Tuning (Lu et al., 2025) stages curriculum complexity progressively, CuES (Mai et al., 2025) generates tasks via intrinsic curiosity, and Dr. Zero (Yue et al., 2026) runs a proposer–solver co-evolution loop without labeled data. Fathoming differs by selecting frontier examples at the tool-call-slice level and re-probing after each GRPO round so the frontier tracks the updated model.

3 Method

Our method turns agent skills from the web into tool-use training signal in two stages, shown in Figure 1. The two stages target the two obstacles to scaling multi-turn tool-use data: producing executable environments at scale, and using the resulting trajectories in a way that matches the target model's current ability. **Grafting** (Section 3.1) is a synthesis loop that builds executable environments from crawled skills and harvests a pool of verified trajectories $\mathcal{T}$ from them (Figure 1a,b). **Fathoming** (Section 3.2) is a training loop that draws on $\mathcal{T}$ at the target model's current capability boundary (Figure 1c). The two stages run in sequence and are connected only by $\mathcal{T}$: Grafting builds the pool once, and Fathoming re-reads it against an evolving model rather than consuming it as a fixed dataset.

3.1 Grafting: Environment Synthesis Loop

Each Grafting stage is run by a dedicated agent (Figure 1b) under a recurring design principle: every agent does open-ended construction but is gated by a validation contract, so the agents supply coverage and variety while the contracts keep their output deterministic and reproducible.

3.1.1 Seed Skills

Grafting seeds each environment from an agent skill (Agent Skills Project, 2026), which supplies the operational structure that bare tool or domain specifications lack: a workflow, executable scripts, and reference material. We crawl public skills, filter out those containing sensitive information, and keep only those that ship a `scripts/` directory, since bundled scripts signal that the workflow can be grounded in a deterministic executable backend (Figure 1a). A kept skill serves only as a workflow-level seed: the synthesis agent reads it to infer operations, state, constraints, and realistic user goals, but the admitted environment is newly generated, with its own tool schemas, database, backend, and tests, and cannot depend on source-repository files at runtime. We release the synthesized environments and trajectories but not the seed skills themselves; see the ethics statement for provenance.

3.1.2 Environment Synthesis

Materializing a seed skill as a portable executable environment is open-ended, requiring the design of tool schemas, a relational world, deterministic backend logic, and tests, so Grafting delegates construction to a coding-agent harness such as Claude Code (Anthropic, 2025) or Codex (OpenAI, 2025) running in an isolated sandbox, and constrains it with a hard output contract (Figure 2). A runner mounts guidance and environment-synthesis skills; the coding agent works through access, design, write, and self-test phases; and the runner admits the environment package only if a deterministic check passes: all required files present and the full test suite green. An admitted environment $e = (\mathcal{S}, \mathcal{R}, \mathcal{W}, \mathcal{Q})$ has four layers that mirror its directory: a specification `spec/` ($\mathcal{S}$: brief, tool schemas, dependency graph), a runtime `runtime/` ($\mathcal{R}$: deterministic backend execution), a world `world/` ($\mathcal{W}$: a pre-populated relational database

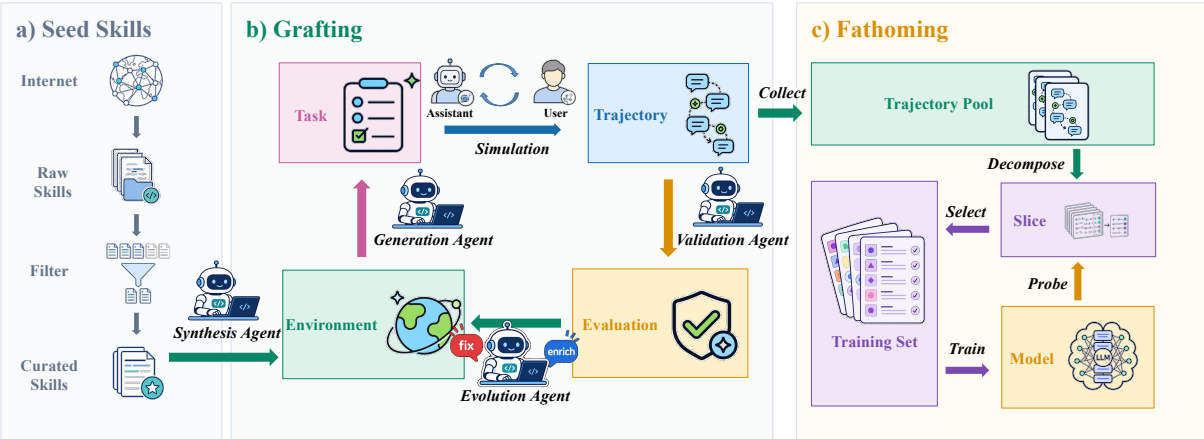

Figure 1: Overview of our method. **(a)** Seed skills are crawled and filtered to a curated pool. **(b) Grafting** is a closed synthesis loop; accepted trajectories form the verified pool $\mathcal{T}$. **(c) Fathoming** decomposes $\mathcal{T}$ into tool-call slices, probes the target model π_{θ_r} to select a frontier subset $\mathcal{B}_r$, trains with GRPO, and re-probes the updated model.

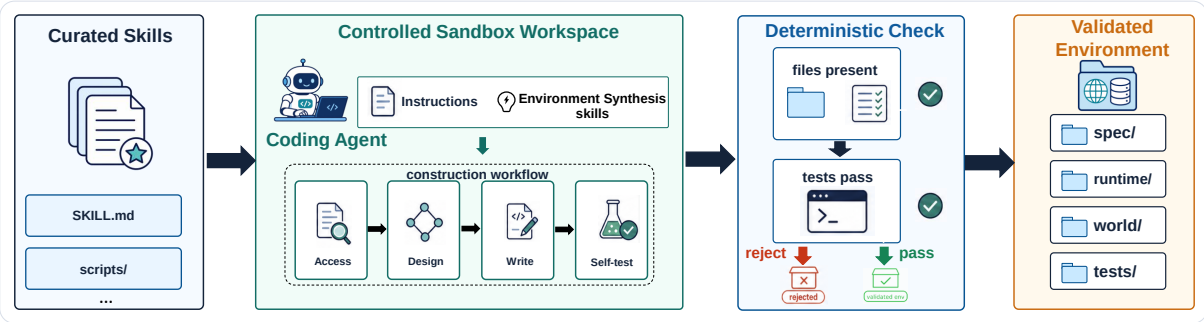

Figure 2: Environment synthesis. A coding agent works inside a controlled sandbox through *access*, *design*, *write*, and *self-test* phases. The environment package is admitted only if a deterministic check passes, yielding a validated environment with `spec/`, `runtime/`, `world/`, and `tests/` layers.

and its schema), and tests `tests/` ($\mathcal{Q}$: an executable suite over tools and workflows). This runner–harness–contract pattern recurs in the later Grafting stages with stage-specific contracts. Appendix A gives the concrete file structure and a running `travel_booking` example.

3.1.3 Task Generation, Simulation, and Validation

With an environment e in hand, Grafting produces trajectories in three steps (Figure 3).

Task generation. A generation agent, constrained by the environment specification, emits grounded task specifications u (each a demand, the database preconditions it requires, a user persona, and an expected tool subgraph) and discards ungrounded, unsolvable, ambiguous, or duplicate tasks before any simulation. The generation agent is also guided by the Python runner, which mounts instructions and task-generation skills. The runner organizes generation through access, design, write, and self-test phases. See Appendices A and H for running examples.

Simulation. A simulation engine runs the user–assistant dialogue on a session-local copy of the world while dispatching assistant tool calls to the deterministic backend. The resulting trajectory τ records user and assistant messages, structured tool calls, backend observations, and the final answer. Appendix I documents the user and assistant agent prompt templates.

Validation. A fluent dialogue is not necessarily a correct one, so validation is a separate step. A validation agent equipped with trajectory-validation skills replays the recorded tool calls on a fresh database and checks that the demand is fulfilled, that assistant claims are grounded in tool results, and that the user stayed consistent with its persona. Accepted trajectories enter the verified pool $\mathcal{T} = \{\tau : v(\tau) = \text{ACCEPTED}\}$ that Fathoming consumes; rejected ones are tagged

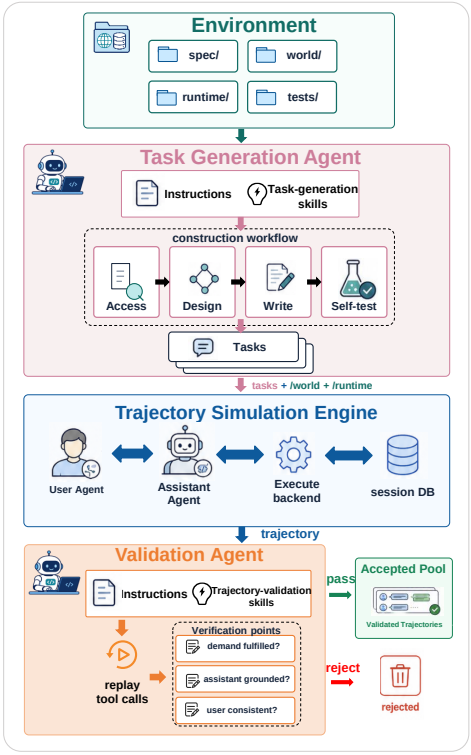

Figure 3: Task generation, simulation, and validation.

with a reason such as `demand_not_fulfilled`, `assistant_hallucination`, or `user_drift`. Appendix J documents the validation prompts and mounted skills.

3.1.4 Environment Evolution

A single synthesis pass leaves environments uneven: some carry latent backend or contract bugs, while others are correct but seeded too uniformly to yield varied trajectories. Environment evolution uses trajectory verdicts as a cheap diagnostic to route each environment and close the Grafting loop (Figure 1b). A smoke-test batch sends each environment to **fix** (internal errors or universal failure), **pass** (a healthy mix of accepted and rejected trajectories), or **enrich** (every trajectory passes, signaling overly uniform seed data). Fix repairs backend or contract inconsistencies; enrich adds rows, edge cases, and distributional variety without changing tool schemas or backend code. Only error-free environments that yield accepted trajectories proceed to large-scale production. This verdict-driven repair is what makes Grafting a loop rather than a one-shot synthesis step, letting environment coverage scale without sacrificing reproducibility. Appendix K documents the evolution prompts and mounted repair and enrichment skills.

3.2 Fathoming: Iterative Training

Grafting's pool $\mathcal{T}$ is correct with respect to the environments, but correctness is not the same as useful training signal for a particular model: a trajectory the model already solves contributes no gradient, and one far beyond it produces no informative gradient either. This distinction is model-specific and, crucially, it moves as the model improves. Fathoming addresses this by selecting training data at the model's current capability boundary and refreshing that selection every round (Figure 1c). It first decomposes each trajectory into the unit at which tool-use signal actually lives: a single held-out tool call in context. For a trajectory τ with available tool set $\mathcal{F}_\tau$ and $N(\tau)$ tool calls, and for each depth $d \in \{1, \ldots, N(\tau)\}$ counted from the end, Fathoming forms a tool-call prediction slice $(\mathcal{F}_\tau, \mathrm{ctx}_{\tau,d}, y_{\tau,d})$, where $\mathrm{ctx}_{\tau,d}$ is the dialogue history before the held-out call (prior user and assistant messages, tool calls, and tool observations), and $y_{\tau,d}$ is the ground-truth tool call. At round r, Fathoming probes π_{θ_r} over these slices, selects a frontier set $\mathcal{B}_r$, trains with GRPO, and re-probes the resulting checkpoint $\pi_{\theta_{r+1}}$.

3.2.1 Capability-Boundary Probing

Probing measures, rather than assumes, how hard each slice is for the current model. At round r we sample π_{θ_r} on each slice $(\mathcal{F}_\tau, \mathrm{ctx}_{\tau,d}, y_{\tau,d})$ $K{=}8$ times at temperature 1.0 and compute a pass rate $p^{(r)}_{\tau,d}$ (order-agnostic match to $y_{\tau,d}$) and a format rate $q^{(r)}_{\tau,d}$ (well-formed tool call). These determine a bucket label:

$$\ell^{(r)}_{\tau,d} = \begin{cases} \texttt{too_easy}, & p^{(r)}_{\tau,d} \geq 7/8, \\ \texttt{effective}, & 0 < p^{(r)}_{\tau,d} < 7/8, \\ \texttt{hard}, & p^{(r)}_{\tau,d} = 0 \wedge q^{(r)}_{\tau,d} > 0, \\ \texttt{too_hard}, & p^{(r)}_{\tau,d} = 0 \wedge q^{(r)}_{\tau,d} = 0. \end{cases} \quad (1)$$

The buckets separate four regimes: `effective` slices sit on the current capability boundary, `too_easy` slices are already solved, and `hard` versus `too_hard` distinguishes a content error (right format, wrong call) from a format failure.

3.2.2 Frontier-Targeted Selection

The buckets rank slices by how much the current model can learn from them. `too_easy` slices carry no signal and are dropped; `too_hard` slices are too far from the model's reach to bridge in one training step and are also dropped. `effective` slices (the

active frontier) are retained in full, and a fraction of `hard` slices is mixed in for aspirational signal. The retained slices form the round's training set $\mathcal{B}_r$.

3.2.3 Iterative GRPO Loop

Dense tool-call reward. Exact-match supervision yields a single bit per slice and discards most of the structure in a tool call. We instead train with GRPO under a dense reward that scores the parts of a predicted call separately. For a sampled response, let F_{fmt} denote valid `<tool_call>` formatting, S_{name} tool-name recall, S_{key} argument-key Jaccard similarity, S_{val} canonicalized argument-value similarity, and $S_{\text{cnt}} = \max(0, 1-|n-m|/\max(m,1))$ the call-count score. With penalties for extra, malformed, and duplicate calls (Appendix C), the dense reward is

$$\begin{aligned}\tilde{R} = {} & 0.10F_{\text{fmt}} + 0.30S_{\text{name}} + 0.25S_{\text{key}} \\ & + 0.25S_{\text{val}} + 0.10S_{\text{cnt}} \\ & - P_{\text{extra}} - P_{\text{invalid}} - P_{\text{dup}},\end{aligned} \tag{2}$$

and the final reward is

$$R = \begin{cases} 1, & \text{canon}(\widehat{\mathbf{y}}) = \text{canon}(\mathbf{y}), \\ \text{clip}_{[0,1]}(0.5\tilde{R}), & \text{otherwise}, \end{cases} \tag{3}$$

where $\widehat{\mathbf{y}}$ and $\mathbf{y}$ are the predicted and ground-truth tool-call sequences. For each selected slice $(\mathcal{F}, \text{ctx}, y) \sim \mathcal{B}_r$, the old policy samples a group of G_{grp} candidate responses $\{O_i\}_{i=1}^{G_{\text{grp}}}$ with rewards $\{R_i\}_{i=1}^{G_{\text{grp}}}$. We optimize

$$\begin{aligned}\mathcal{J}_{\text{GRPO}}(\theta) &= \mathop{\mathbb{E}}_{\mathcal{B}_r, \pi_{\text{old}}} \left[\bar{L}\right], \\ \bar{L} &= \frac{1}{G_{\text{grp}}} \sum_{i=1}^{G_{\text{grp}}} \left(L_i - \beta D_{\text{KL}}\right),\end{aligned} \tag{4}$$

where $\pi_{\text{old}} = \pi_{\theta_{\text{old}}}$, $(\mathcal{F}, \text{ctx}, y) \sim \mathcal{B}_r$, and $\{O_i\}_{i=1}^{G_{\text{grp}}} \sim \pi_{\text{old}}(\cdot \mid \text{ctx}, \mathcal{F})$. The ratio and group-normalized advantage are

$$\begin{aligned}L_i &= \min\big(\rho_i A_i, \\ &\qquad \text{clip}(\rho_i, 1-\epsilon, 1+\epsilon)A_i\big), \\ D_{\text{KL}} &= \text{KL}(\pi_\theta \,\|\, \pi_{\text{old}}), \\ \pi_i &= \pi_\theta(O_i \mid \text{ctx}, \mathcal{F}), \\ \pi_i^{\text{old}} &= \pi_{\text{old}}(O_i \mid \text{ctx}, \mathcal{F}), \\ \rho_i &= \pi_i / \pi_i^{\text{old}}, \\ A_i &= \frac{R_i - \text{mean}(\{R_j\}_{j=1}^{G_{\text{grp}}})}{\text{std}(\{R_j\}_{j=1}^{G_{\text{grp}}})}.\end{aligned} \tag{5}$$

Re-probing. After a round produces a checkpoint, Fathoming re-probes it against the same trajectory pool, recomputes bucket assignments, reassembles frontier-targeted data, and begins the next GRPO round from the previous checkpoint. Because the frontier shifts as the model improves (slices that were `effective` become `too_easy`), each round concentrates training compute on signal that is informative for the *current* model rather than for a fixed difficulty distribution. Section 4.1 reports two such rounds on Qwen3-4B-Instruct-2507.

Together, Grafting and Fathoming turn crawled developer skills into capability-matched reinforcement signal, with neither hand-authored environments nor a static difficulty distribution.

4 Experiments

4.1 Experimental Setup

Environment and trajectory synthesis. All training data comes from an initial pipeline-development run of Grafting (Section 3.1) over 500 source environments, yielding 12,745 verified multi-turn trajectories (Appendix B). We separately scale synthesis to produce GROVE-5K, a release pool of 5,101 environments characterized in Section 4.4; GROVE-5K is a synthesis artifact only and is *not* used for the training experiments here.

Fathoming training. We instantiate Fathoming (Section 3.2) on Qwen3-4B-Instruct-2507. Each round probes the current checkpoint at temperature 1.0 with K=8 samples per trajectory slice, retains all `effective` slices and 10% of `hard` slices, and trains with GRPO under the dense tool-call reward of Section 3.2.3. We run two probe–train rounds: round 1 selects 7,388 samples from the pool; round 2 starts from the round-1 checkpoint, re-probes the same pool, and selects 3,616 samples. Additional training configuration is in Appendix C.

Benchmarks. We evaluate every checkpoint on two benchmarks; neither is represented in training, so both constitute out-of-distribution evaluation. BFCLv4 (Patil et al., 2025) multi-turn reports overall accuracy and four sub-categories (Base, Missing Function, Missing Parameter, Long Context). ComplexFuncBench (Zhong et al., 2025) reports per-call accuracy, per-task success rate, and five domain splits (Car-Rental, Flight, Hotel, Attraction, Cross-Domain). Both target multi-turn interaction

Model	Overall	Base	Miss F.	Miss P.	Long
Closed-source models					
Claude-Opus-4.5 (FC)	68.38	81.0	64.0	58.0	70.5
Gemini-3-Pro (FC)	63.12	69.0	63.0	56.5	64.0
Claude-Sonnet-4.5 (FC)	61.37	69.0	65.0	52.5	59.0
GPT-4.1 (FC)	38.88	47.5	32.5	32.5	43.0
Open-source models					
GLM-4.6 (FC thinking)	68.00	74.5	68.0	63.0	66.5
Qwen3-8B (FC)	41.75	50.5	42.0	40.0	34.5
xLAM-2-3b-fc-r (FC)	58.38	71.5	59.0	57.5	45.5
Nanbeige4-3B-Thinking-2511 (FC)	51.12	58.5	54.0	45.0	47.0
Arch-Agent-3B	34.88	42.0	37.5	31.0	29.0
Hammer2.1-3b (FC)	16.50	22.0	12.5	16.0	15.5
Ours					
Qwen3-4B-Inst-2507 (official)	22.12	26.5	21.0	15.5	25.5
Qwen3-4B-Inst-2507	22.00	26.30	20.70	16.60	24.40
FATHOM-4B	39.70	56.80	27.60	27.90	46.50

Table 1: Main BFCLv4 multi-turn results. Leaderboard scores are from the official BFCLv4 website (Berkeley Function Calling Leaderboard, 2026) (2026-04-12 snapshot). Our local BFCLv4 scores report means over five evaluation runs, omitting standard deviations for compactness. BFCLv4 is not represented in training.

Model	Call	Succ.	Car	Flight	Hotel	Attr.	Cross
Closed-source models							
GPT-4o	80.55	60.50	86.88	85.50	81.99	87.59	75.13
Claude-3.5-Sonnet	79.27	61.00	86.01	79.50	68.17	83.33	79.33
Open-source models							
Qwen2.5-7B	18.19	5.00	6.41	5.13	20.65	35.18	21.41
GLM-4-9B	27.97	9.40	25.36	34.00	31.52	29.26	25.46
Mistral Large 2	48.78	20.10	58.16	52.88	34.32	40.18	50.54
Qwen2.5-72B	58.32	40.10	57.58	49.25	60.24	67.41	59.00
Ours							
Qwen3-4B-Inst-2507	49.62	0.9	69.5	54.6	30.0	43.9	48.8
FATHOM-4B	**61.64**	0.7	80.8	61.1	35.1	54.3	64.9

Table 2: Main ComplexFuncBench results. External-model rows are from Table 2 of Zhong et al. (2025); columns report overall call accuracy, task success rate, and domain-level call accuracy. Bold marks the best open-source overall call accuracy in the comparison. ComplexFuncBench is not represented in training.

and tool use, matching our data-construction setting.

4.2 Results

BFCLv4 multi-turn. Table 1 shows that Fathoming lifts Qwen3-4B-Instruct-2507 from $22.00_{\pm 0.20}$ to $39.70_{\pm 0.52}$ (+17.7 points), surpassing GPT-4.1 (38.88) and nearly matching Qwen3-8B (41.75) at half the parameter count. The largest gains appear on Base (+30.5) and Long Context (+22.1).

ComplexFuncBench. Table 2 shows that per-call accuracy improves from 49.62 to 61.64 (+12.0), surpassing all open-source baselines reported by ComplexFuncBench, including Qwen2.5-72B (58.32), a model 18× its size, and Mistral Large 2 (48.78). The improvement holds across all five domains, with the largest gains on Cross-Domain (+16.1), Car-Rental (+11.3), and Attraction (+10.4). Per-task success rate remains near the floor (0.9% → 0.7%), expected for a 4B model on long-context multi-step tasks where a single incorrect call fails the entire task; thus, the result is best read as a call-level grounding improvement rather than full task completion.

Variant	Ovr.	Base	Miss F.	Miss P.	Long
Training signal					
Base model	22.00	26.30	20.70	16.60	24.40
Random-depth GRPO	21.65	24.50	21.40	18.10	22.60
Last-depth GRPO	24.28	29.40	20.10	23.00	24.60
Frontier SFT (R1)	26.00	30.70	19.00	25.80	28.50
Frontier GRPO (R1)	36.45	48.20	30.70	26.40	40.50
Iterative re-probing					
Round 1	36.45	48.20	30.70	26.40	40.50
Round 2	39.70	56.80	27.60	27.90	46.50
Source-environment scaling					
0 env.	22.00	26.30	20.70	16.60	24.40
50 env.	23.65	25.10	23.80	21.70	24.00
500 env.	39.70	56.80	27.60	27.90	46.50

Table 3: Ablations on BFCLv4 multi-turn accuracy. Columns report Overall, Base, Missing Function, Missing Parameter, and Long Context; higher is better. Entries are five-run means. Trained variants in the training-signal block use the round-1 sample budget. Repeated rows are anchors, not separate runs: Base model equals 0 env., Frontier GRPO (R1) equals Round 1, and Round 2 equals the 500 env. setting. Standard deviations are omitted for compactness.

4.3 Ablations

Table 3 reports BFCLv4 multi-turn ablations for training signal, iterative re-probing, and source-environment scaling.

Training signal. The training-signal block isolates frontier-targeted selection and dense-reward GRPO. Random-depth and last-depth GRPO barely improve over the base model, and frontier SFT is much weaker than Frontier GRPO, indicating that the gains require both model-aware selection and reinforcement learning rather than tool-call data alone.

Iterative re-probing. Round 2 adds gains over Round 1, especially on Base and Long Context, suggesting that refreshing the frontier after the model improves can help. The Missing Function drop suggests a trade-off: continued positive-only optimization may encourage the model to attempt calls more often, sometimes at the expense of correctly declining unavailable functions. Appendix D analyzes how the probing distribution shifts between rounds.

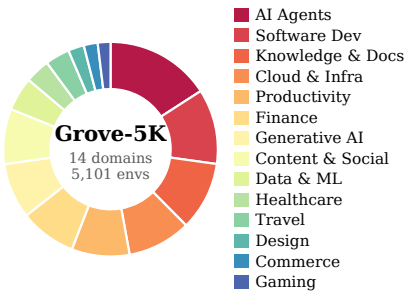

Figure 4: Domain taxonomy of GROVE-5K: the 14 first-tier domains. Wedge area is proportional to the number of environments in each domain.

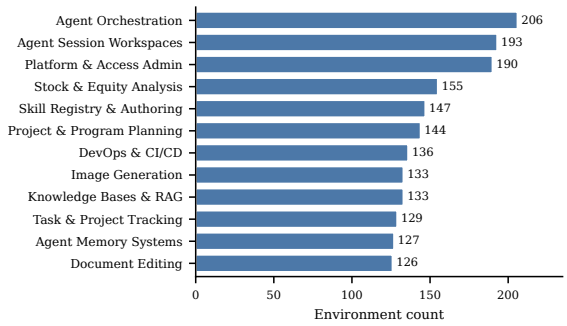

Figure 5: The top-12 GROVE-5K sub-domains by environment count.

Source-environment scaling. Scaling from 50 to 500 source environments changes the result from marginal improvement over the base model to the main gain, suggesting that Fathoming benefits from broader source coverage.

4.4 Environment Analysis

Scaling to GROVE-5K. Beyond the 500-environment pipeline-development run used for training, the same synthesis loop (Section 3.1.2) produces GROVE-5K, a release pool of 5,101 portable environments. The median package exposes 25 tools (range 14–57), defines 9 relational tables (range 3–27), and implements its backend in 887 lines of deterministic Python (range 450–2,478).

Domain distribution. Embedding-based clustering followed by manual labeling groups the 5,101 environments into 14 first-tier domains and over 50 sub-domains (Figures 4 and 5). The main takeaway is breadth rather than a specific domain ranking: the largest domain, *AI Agents & Autonomy*, accounts for only 15.9% of the pool, while the remaining environments span software, knowledge and document work, infrastructure, finance, media, healthcare, and games. The leading sub-domains likewise mix orchestration, analysis, DevOps, generation, and editing workflows, suggesting that GROVE-5K is not concentrated in a single templated source family.

Tool diversity. Embedding the 130,985 tools and counting connected components in a top-50 cosine-similarity graph yields 83,350 *effective tools*, while the Vendi score (Friedman and Dieng, 2023) saturates near 118 effective dimensions, far below the 1,536-dimensional ambient ceiling. This suggests near-linear growth in distinct functions despite a low-dimensional geometric spread; Appendix L reports the full scaling analysis.

5 Conclusion

We presented a methodology for growing executable environments and training tool-use agents from web-published developer skills. First, Grafting turns filtered skills into self-contained executable environments through coding-agent synthesis under deterministic validation contracts. These environments support grounded task and trajectory synthesis, yielding verified multi-turn training data; at scale, the same synthesis loop produces GROVE-5K, a pool of 5,101 portable environments with broad domain and tool diversity. Second, Fathoming shows how to train effectively on these synthesized trajectories by repeatedly probing the target model, selecting frontier-near verified examples, and applying GRPO. On trajectories from an initial 500-environment run, this iterative procedure produces FATHOM-4B and improves Qwen3-4B-Instruct-2507 by 17.7 points on BFCLv4 multi-turn and 12.0 points on ComplexFuncBench per-call accuracy after two probe–train rounds. Together, validation-driven synthesis and capability-matched iterative training make skill-derived environments a reliable substrate for advancing tool-use agents.

Limitations

Scale gap between synthesis and training. Our end-to-end Fathoming experiment was conducted on an initial 500 environments produced during pipeline development, yielding approximately 12,000 verified trajectories for training. We subsequently scaled Grafting to 5,101 environments (GROVE-5K) but have not yet run trajectory generation, probing, or GRPO training over the full pool due to compute cost (Appendix F). The diversity

analysis in Section 4.4 and Appendix L (Figure 6) characterizes the GROVE-5K artifacts intrinsically; whether scaling the training pool would yield further downstream gains is left to future work.

Automated validation of synthetic artifacts. Grafting validates environments, tasks, and trajectories through deterministic checks, replay against fresh state, and a separate trajectory-validation agent. These contracts establish internal consistency with the generated backend and task specification, but they do not constitute an independent human audit of artifact realism, task usefulness, or judge reliability.

Positive-only training signal. Fathoming slices accepted trajectories at tool-call positions, so the main training objective directly rewards correct tool-call prediction. Missing-function and missing-parameter scenarios may appear in dialogue context, but refusal and clarification turns are not optimized as no-call response targets. This one-sided signal likely explains the Missing Function drop from round 1 to round 2 in Table 3: the model is trained to predict tool calls, biasing it toward attempting calls even when the appropriate action is to refuse. A pilot experiment augmenting the round-2 training set with 300 refusal and clarification response targets recovers most of the Missing Function gap ($+4.1$) but incurs a substantial cost on Base (-5.7) and Long Context (-5.8), yielding a net overall decrease (Appendix E). Resolving this trade-off—through dedicated refusal-training stages, ratio tuning between tool-call and no-call targets, or reward shaping—remains an open direction.

Single target model. We evaluate Fathoming on a single model (Qwen3-4B-Instruct-2507) and do not test alternative model families, larger scales, or reasoning-oriented models. The probing thresholds (e.g., $\geq 7/8$ for `too_easy`) and the 10% hard-sample inclusion rate were set without cross-model tuning; these hyperparameters may require adjustment for models of different capacity or architecture.

Ethics Statement

Data sourcing, licensing, and release. The skills used as seed material were crawled from public GitHub repositories via skill-aggregator indices, and their source repositories have heterogeneous licensing signals. We therefore treat source provenance as a release-governance constraint rather than assuming that raw skills are redistributable. We do not redistribute the raw skill collection as a training artifact. Grafting uses each skill as a workflow-level specification seed: the synthesis worker may read `SKILL.md`, scripts, and metadata to infer operations, state, constraints, and realistic user goals, but each admitted environment must contain newly generated tool schemas, backend implementation, database schema and seed data, documentation, and tests. The released environments therefore do not include raw skill files and cannot depend on source-repository files at runtime. This mitigates provenance risk, but we do not treat source licensing as irrelevant. For public GROVE-5K releases, we will provide source-repository URLs and license metadata when available, avoid redistributing raw skills, and support filtering or removal of artifacts associated with incompatible source terms or maintainer requests.

Use of AI assistants. AI assistants were used in two distinct roles. First, as part of the research method, coding-agent harnesses such as Claude Code and Codex were invoked to synthesize environments, tasks, and trajectories, and to validate them under the contracts described in Section 3 and Appendix G–K. These uses are experimental procedures rather than authorship contributions. Second, AI writing assistants were used during manuscript preparation for drafting, editing, and polishing text. The human authors selected the research questions, designed the experiments, inspected and interpreted the results, decided what artifacts to release, and take responsibility for all claims, analyses, and final manuscript content. No AI assistant is listed as an author.

References

Agent Skills Project. 2026. Agent Skills Specification. https://agentskills.io/specification. Repository: https://github.com/agentskills/agentskills. Accessed 2026-05-26.

Anthropic. 2025. Claude Code. https://www.anthropic.com/claude-code. Agentic coding tool.

Berkeley Function Calling Leaderboard. 2026. Berkeley Function Calling Leaderboard (BFCL) V4. https://gorilla.cs.berkeley.edu/leaderboard.html. Official leaderboard snapshot dated 2026-04-12.

Shihao Cai, Runnan Fang, Jialong Wu, Baixuan Li, Xinyu Wang, Yong Jiang, Liangcai Su, Liwen Zhang, Wenbiao Yin, Zhen Zhang, Fuli Feng, Pengjun Xie, and Xiaobin Wang. 2025. AutoForge: Automated Environment Synthesis for Agentic Reinforcement Learning. *Preprint*, arXiv:2512.22857.

Mingyue Cheng, Jie Ouyang, Shuo Yu, Ruiran Yan, Yucong Luo, Zirui Liu, Daoyu Wang, Qi Liu, and Enhong Chen. 2025. Agent-R1: Training Powerful LLM Agents with End-to-End Reinforcement Learning. *Preprint*, arXiv:2511.14460.

Guanting Dong, Yifei Chen, Xiaoxi Li, Jiajie Jin, Hongjin Qian, Yutao Zhu, Hangyu Mao, Guorui Zhou, Zhicheng Dou, and Ji-Rong Wen. 2025. Tool-Star: Empowering LLM-Brained Multi-Tool Reasoner via Reinforcement Learning. *Preprint*, arXiv:2505.16410.

Guanting Dong, Junting Lu, Junjie Huang, Wanjun Zhong, Longxiang Liu, Shijue Huang, Zhenyu Li, Yang Zhao, Xiaoshuai Song, Xiaoxi Li, Jiajie Jin, Yutao Zhu, Hanbin Wang, Fangyu Lei, Qinyu Luo, Mingyang Chen, Zehui Chen, Jiazhan Feng, Ji-Rong Wen, and Zhicheng Dou. 2026. AgentWorld: Scaling Real-World Environment Synthesis for Evolving General Agent Intelligence. *Preprint*, arXiv:2604.18292.

Runnan Fang, Shihao Cai, Baixuan Li, Jialong Wu, Guangyu Li, Wenbiao Yin, Xinyu Wang, Xiaobin Wang, Liangcai Su, Zhen Zhang, Shibin Wu, Zhengwei Tao, Yong Jiang, Pengjun Xie, Fei Huang, and Jingren Zhou. 2025. Towards General Agentic Intelligence via Environment Scaling. *Preprint*, arXiv:2509.13311.

Jiazhan Feng, Shijue Huang, Xingwei Qu, Ge Zhang, Yujia Qin, Baoquan Zhong, Chengquan Jiang, Jinxin Chi, and Wanjun Zhong. 2025. ReTool: Reinforcement Learning for Strategic Tool Use in LLMs. *Preprint*, arXiv:2504.11536.

Dan Friedman and Adji Bousso Dieng. 2023. The Vendi score: A diversity evaluation metric for machine learning. *Transactions on Machine Learning Research*. ArXiv:2210.02410.

Jiacheng Guo, Ling Yang, Peter Chen, Qixin Xiao, Yinjie Wang, Xinzhe Juan, Jiahao Qiu, Ke Shen, and Mengdi Wang. 2025. GenEnv: Difficulty-Aligned Co-Evolution Between LLM Agents and Environment Simulators. *Preprint*, arXiv:2512.19682.

Zhicheng Guo, Sijie Cheng, Hao Wang, Shihao Liang, Yujia Qin, Peng Li, Zhiyuan Liu, Maosong Sun, and Yang Liu. 2024. StableToolBench: Towards Stable Large-Scale Benchmarking on Tool Learning of Large Language Models. *Preprint*, arXiv:2403.07714.

Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2021. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547.

Xuefeng Li, Haoyang Zou, and Pengfei Liu. 2025a. ToRL: Scaling Tool-Integrated RL. *Preprint*, arXiv:2503.23383.

Yuetai Li, Huseyin A Inan, Xiang Yue, Wei-Ning Chen, Lukas Wutschitz, Janardhan Kulkarni, Radha Poovendran, Robert Sim, and Saravan Rajmohan. 2025b. Simulating Environments with Reasoning Models for Agent Training. *Preprint*, arXiv:2511.01824.

Weiwen Liu, Xu Huang, Xingshan Zeng, Xinlong Hao, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Zhengying Liu, Yuanqing Yu, Zezhong Wang, Yuxian Wang, Wu Ning, Yutai Hou, Bin Wang, Chuhan Wu, Xinzhi Wang, Yong Liu, Yasheng Wang, and 8 others. 2024a. ToolACE: Winning the Points of LLM Function Calling. *Preprint*, arXiv:2409.00920.

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

Siyuan Lu, Zechuan Wang, Hongxuan Zhang, Qintong Wu, Leilei Gan, Chenyi Zhuang, Jinjie Gu, and Tao Lin. 2025. Don't Just Fine-tune the Agent, Tune the Environment. *Preprint*, arXiv:2510.10197.

Shinji Mai, Yunpeng Zhai, Ziqian Chen, Cheng Chen, Anni Zou, Shuchang Tao, Zhaoyang Liu, and Bolin Ding. 2025. CuES: A Curiosity-driven and Environment-grounded Synthesis Framework for Agentic RL. *Preprint*, arXiv:2512.01311.

OpenAI. 2025. OpenAI Codex. `https://openai.com/index/introducing-codex/`. Agentic coding tool.

OpenAI. 2026. text-embedding-3-small Model Documentation. `https://platform.openai.com/docs/models/text-embedding-3-small`. OpenAI API documentation. Accessed 2026-05-26.

Shishir G. Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. 2025. The berkeley function calling leaderboard (BFCL): From tool use to agentic evaluation of large language models. In *Forty-second International Conference on Machine Learning*.

Akshara Prabhakar, Zuxin Liu, Ming Zhu, Jianguo Zhang, Tulika Awalgaonkar, Shiyu Wang, Zhiwei Liu, Haolin Chen, Thai Hoang, Juan Carlos Niebles, Shelby Heinecke, Weiran Yao, Huan Wang, Silvio Savarese, and Caiming Xiong. 2025. APIGen-MT: Agentic Pipeline for Multi-Turn Data Generation via Simulated Agent-Human Interplay. *Preprint*, arXiv:2504.03601.

Cheng Qian, Emre Can Acikgoz, Qi He, Hongru Wang, Xiusi Chen, Dilek Hakkani-Tür, Gokhan Tur, and Heng Ji. 2025. ToolRL: Reward is All Tool Learning Needs. *Preprint*, arXiv:2504.13958.

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

Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji rong Wen. 2025. Tool learning with large language models: a survey. *Frontiers of Computer Science*, 19(8):198343.

Qwen Team. 2025. Qwen3 Technical Report. *Preprint*, arXiv:2505.09388.

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

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models. *Preprint*, arXiv:2402.03300.

Xiaoshuai Song, Haofei Chang, Guanting Dong, Yutao Zhu, Ji-Rong Wen, and Zhicheng Dou. 2026. EnvScaler: Scaling Tool-Interactive Environments for LLM Agent via Programmatic Synthesis. *Preprint*, arXiv:2601.05808.

Dunwei Tu, Hongyan Hao, Hansi Yang, Yihao Chen, Yi-Kai Zhang, Zhikang Xia, Yu Yang, Yueqing Sun, Xingchen Liu, Furao Shen, Qi Gu, Hui Su, and Xunliang Cai. 2026. ScaleEnv: Scaling Environment Synthesis from Scratch for Generalist Interactive Tool-Use Agent Training. *Preprint*, arXiv:2602.06820.

Zhaoyang Wang, Canwen Xu, Boyi Liu, Yite Wang, Siwei Han, Zhewei Yao, Huaxiu Yao, and Yuxiong He. 2026. Agent World Model: Infinity Synthetic Environments for Agentic Reinforcement Learning. *Preprint*, arXiv:2602.10090.

Zhangchen Xu, Adriana Meza Soria, Shawn Tan, Anurag Roy, Ashish Sunil Agrawal, Radha Poovendran, and Rameswar Panda. 2025. TOUCAN: Synthesizing 1.5M Tool-Agentic Data from Real-World MCP Environments. *Preprint*, arXiv:2510.01179.

Chen Yang, Ran Le, Yun Xing, Zhenwei An, Zongchao Chen, Wayne Xin Zhao, Yang Song, and Tao Zhang. 2025. ToolMind Technical Report: A Large-Scale, Reasoning-Enhanced Tool-Use Dataset. *Preprint*, arXiv:2511.15718.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. 2024. τ-bench: A Benchmark for Tool-Agent-User Interaction in Real-World Domains. *Preprint*, arXiv:2406.12045.

Fan Yin, Zifeng Wang, I-Hung Hsu, Jun Yan, Ke Jiang, Yanfei Chen, Jindong Gu, Long T. Le, Kai-Wei Chang, Chen-Yu Lee, Hamid Palangi, and Tomas Pfister. 2025. Magnet: Multi-turn Tool-use Data Synthesis and Distillation via Graph Translation. *Preprint*, arXiv:2503.07826.

Zhenrui Yue, Kartikeya Upasani, Xianjun Yang, Suyu Ge, Shaoliang Nie, Yuning Mao, Zhe Liu, and Dong Wang. 2026. Dr. Zero: Self-Evolving Search Agents without Training Data. *Preprint*, arXiv:2601.07055.

Xingshan Zeng, Weiwen Liu, Xu Huang, Zezhong Wang, Lingzhi Wang, Liangyou Li, Yasheng Wang, Lifeng Shang, Xin Jiang, Ruiming Tang, and Qun Liu. 2025a. ToolACE-R: Model-aware Iterative Training and Adaptive Refinement for Tool Learning. *Preprint*, arXiv:2504.01400.

Yirong Zeng, Xiao Ding, Yutai Hou, Yuxian Wang, Li Du, Juyi Dai, Qiuyang Ding, Duyu Tang, Dandan Tu, Weiwen Liu, Bing Qin, and Ting Liu. 2025b. Tool Zero: Training Tool-Augmented LLMs via Pure RL from Scratch. *Preprint*, arXiv:2511.01934.

Kangning Zhang, Wenxiang Jiao, Kounianhua Du, Yuan Lu, Weiwen Liu, Weinan Zhang, and Yong Yu. 2025. LoopTool: Closing the Data-Training Loop for Robust LLM Tool Calls. *Preprint*, arXiv:2511.09148.

Lucen Zhong, Zhengxiao Du, Xiaohan Zhang, Haiyi Hu, and Jie Tang. 2025. ComplexFuncBench: Exploring Multi-Step and Constrained Function Calling under Long-Context Scenario. *Preprint*, arXiv:2501.10132.

A Environment, Task, and Trajectory Examples

This appendix illustrates the concrete artifacts produced by Grafting, using the `travel_booking` environment from GROVE-5K as a running example. The same package layout is described in Section 3.1.2.

A.1 Environment Package Structure

Each synthesized environment is a self-contained directory with the following structure:

```
<env_name>/
  spec/
    ENVIRONMENT.md       # brief
    tools.jsonl          # tools
    tool_graph.json      # tool graph
  runtime/
    backend.py           # backend
  tests/
    test_runtime.py      # pytest
  world/
    schema.sql           # schema
    world.db             # seeded DB
```

Specification layer. The specification layer is the interface seen by task generation, simulation, and validation. `ENVIRONMENT.md` gives a natural-language summary of the domain, tool inventory, authentication rules, workflows, schema tables, and verification hints. `tools.jsonl` defines every tool exposed by the environment, one JSON object per line, specifying the tool name, natural-language description, category, and input schema. `tool_graph.json` encodes directed edges between tools, capturing state preconditions and data-flow prerequisites (e.g., `authenticate_travel` must precede `book_flight` because it provides an `access_token`). The `travel_booking` example exposes 19 tools; the following abridged excerpts show one mutation tool and two graph edges:

```
{"name": "book_flight", "category": "mutation",
 "description": "Book a flight between two airports
   and charge the specified credit card.",
 "inputSchema": {"type": "object",
   "properties": {
     "access_token": {"type": "string"},
     "card_id": {"type": "string"},
     "travel_date": {"type": "string"},
     "travel_from": {"type": "string"},
     "travel_to": {"type": "string"},
     "travel_class": {"type": "string",
      "enum": ["economy", "business", "first"]}},
   "required": ["access_token", "card_id",
     "travel_date", "travel_from", "travel_to",
     "travel_class"]}}

{"from": "authenticate_travel",
 "to": "book_flight",
 "type": "state_precondition",
 "via": "access_token"}
{"from": "get_flight_cost",
 "to": "book_flight",
 "type": "data_dependency",
 "via": "travel_cost"}
```

World layer. `schema.sql` defines the relational schema and `world.db` provides a pre-populated SQLite database with realistic seed data. Each task's simulation operates on a session-local copy of `world.db`, ensuring isolation between tasks while preserving persistent state within a trajectory. The following excerpt shows a representative subset of the `travel_booking` schema:

```
CREATE TABLE airports (
    iata_code TEXT PRIMARY KEY,
    city_name TEXT NOT NULL
);
CREATE TABLE flight_routes (
    origin      TEXT NOT NULL
                REFERENCES airports(iata_code),
    destination TEXT NOT NULL
                REFERENCES airports(iata_code),
    base_cost_usd INTEGER NOT NULL,
    PRIMARY KEY (origin, destination)
);
CREATE TABLE session (
   session_id      INTEGER PRIMARY KEY DEFAULT 1,
    access_token     TEXT,
    token_type       TEXT,
    token_expires_at TEXT,
    token_scope      TEXT,
    user_first_name  TEXT,
    user_last_name   TEXT,
    budget_limit     REAL
);
CREATE TABLE bookings (
    booking_id   TEXT PRIMARY KEY,
    card_id      TEXT NOT NULL
                REFERENCES credit_cards(card_id),
    travel_date  TEXT NOT NULL,
    travel_from  TEXT NOT NULL
                 REFERENCES airports(iata_code),
    travel_to    TEXT NOT NULL
                 REFERENCES airports(iata_code),
    travel_class TEXT NOT NULL
      CHECK(travel_class IN
            ('economy','business','first')),
    travel_cost    REAL NOT NULL,
    transaction_id TEXT NOT NULL
);
```

Runtime layer. `backend.py` implements a Python class that dispatches tool calls against a session-local SQLite database. All tool implementations are deterministic: no network access, no external API keys, no random number generation. The backend exposes a uniform `call(tool_name, arguments) -> result` interface consumed by the simulation harness. Every response uses a unified envelope: `{"success": true, "data": ...}` on success and `{"success": false, "error": {"code": ..., "message": ...}}` on failure.

```
class EnvironmentBackend:
  def __init__(self, *, db_path):
      self._conn = sqlite3.connect(str(db_path))
      self._conn.row_factory = sqlite3.Row
     self._conn.execute("PRAGMA foreign_keys=ON")

  def call(self, tool_name, arguments):
      """Dispatch tool_name to its handler."""
      if tool_name not in self._TOOL_NAMES:
          return self._error("UNKNOWN_TOOL",
              f"Unknown tool: {tool_name}")
     method = getattr(self, f"_tool_{tool_name}")
      return method(**arguments)

  # --- Example tool: book_flight (abridged) ---
  def _tool_book_flight(self, access_token,
          card_id, travel_date,
          travel_from, travel_to, travel_class):
      err = self._check_access(access_token)
      if err:
          return err
      route = self._conn.execute(
        "SELECT base_cost_usd FROM flight_routes "
          "WHERE origin=? AND destination=?",
          (travel_from, travel_to)).fetchone()
      if route is None:
          return self._error("NO_ROUTE",
              "No route for the given airports.")
      cost = self._compute_cost(
          route["base_cost_usd"],
          travel_class, travel_date)
      card = self._conn.execute(
          "SELECT balance FROM credit_cards "
        "WHERE card_id=?", (card_id,)).fetchone()
      if card["balance"] < cost:
         return self._error("INSUFFICIENT_FUNDS",
              f"Required: {cost}, "
              f"available: {card['balance']}.")
      self._conn.execute(
          "UPDATE credit_cards "
         "SET balance=balance-? WHERE card_id=?",
          (cost, card_id))
      booking_id = self._next_id(
          "bookings","booking_id","BK",7)
      self._conn.execute(
        "INSERT INTO bookings (...) VALUES (...)",
          (booking_id, card_id, travel_date,
           travel_from, travel_to,
           travel_class, cost, txn_id))
      self._conn.commit()
      return self._ok({"booking_id": booking_id,
                       "transaction_id": txn_id})
```

Test layer. `test_runtime.py` is a pytest suite that dynamically loads `backend.py`, instantiates the backend against a copied `world.db`, and asserts correctness on every individual tool and on representative multi-step workflows drawn from `tool_graph.json`. A package is admitted only when the full test suite passes. The following workflow test exercises a complete booking lifecycle:

```
def test_full_booking_workflow(auth_backend):
    # 1. Register a new credit card.
    reg = auth_backend.call(
        "register_credit_card",
        {"access_token": TOKEN,
         "card_number": "5555000011112222",
         "expiration_date": "06/30",
         "cardholder_name": "Test User",
         "card_verification_number": 456})
    assert reg["success"] is True
    card_id = reg["data"]["card_id"]
    # 2. Get flight cost.
    cost = auth_backend.call(
        "get_flight_cost",
       {"travel_from": "SFO","travel_to": "LAX",
         "travel_date": "2024-01-15",
         "travel_class": "economy"})
    assert cost["data"]["cost"] == 200.0
    # 3. Book flight.
    book = auth_backend.call(
        "book_flight",
      {"access_token": TOKEN,"card_id": card_id,
         "travel_date": "2024-01-15",
         "travel_from": "SFO","travel_to": "LAX",
         "travel_class": "economy"})
    assert book["success"] is True
    booking_id = book["data"]["booking_id"]
    # 4. Retrieve invoice.
    inv = auth_backend.call("retrieve_invoice",
        {"access_token": TOKEN,
         "booking_id": booking_id})
    assert inv["data"]["travel_cost"] == 200.0
    # 5. Cancel and verify refund.
    cancel = auth_backend.call("cancel_booking",
        {"access_token": TOKEN,
         "booking_id": booking_id})
  assert cancel["data"]["refund_amount"] == 200.0
    bal = auth_backend.call(
        "get_credit_card_balance",
        {"access_token": TOKEN,
         "card_id": card_id})
    assert bal["data"]["balance"] == 10000.0
```

A.2 Task Structure

Each generated task is a JSON object containing: (1) a natural-language demand describing the user's goal; (2) `world_setup` SQL statements that establish task-specific preconditions; (3) a `user_agent_config` specifying role, personality, and knowledge boundary; and (4) a `rule` drawn from the five modes described in Section 3.1.3. The tool-subgraph constraint is injected into the worker's entry prompt and recorded by the task-generation runner, but it is not part of the admitted `task.json` artifact.

```
{
  "demand": "Book a business-class flight
    from SFO to JFK and purchase travel
    insurance for the booking.",
  "world_setup": [
    "UPDATE credit_cards SET balance=50000
     WHERE card_id='CC0000001'"
  ],
  "user_agent_config": {
    "role": "A frequent business traveler",
    "personality": "Polite but time-pressed;
      prefers concise confirmations.",
    "knowledge_boundary":
      "Knows: name is John Doe,
       card id is CC0000001,
       travel date is 2024-03-20.
```

```
      Does not know: booking id, cost,
      or insurance policy id."
  },
  "rule": "normal"
}
```

A.3 Trajectory and Validation Structure

Each trajectory is a sequence of conversation turns alternating between user and assistant messages. Assistant turns may contain one or more tool calls, each recorded as a structured object with `tool_name`, `arguments`, and the `result` returned by the deterministic backend. The following excerpt shows three representative turns:

```
[User] I'm John Doe. I need a business-class
  flight from SFO to JFK on March 20th.
  My card is CC0000001.

[Assistant]
  <tool_call>
  {"name": "authenticate_travel",
   "arguments": {"first_name": "John",
                 "last_name": "Doe"}}
  </tool_call>

[Tool Result]
  {"success": true,
   "data": {"access_token": "token_travel_001",
            "token_type": "Bearer"}}

[Assistant]
  <tool_call>
  {"name": "get_flight_cost",
   "arguments": {"travel_from": "SFO",
     "travel_to": "JFK",
     "travel_date": "2024-03-20",
     "travel_class": "business"}}
  </tool_call>

[Tool Result]
  {"success": true,
   "data": {"cost": 1600.0,
            "currency": "USD",
            "route": "SFO->JFK",
            "class": "business"}}

[Assistant]
  <tool_call>
  {"name": "book_flight",
   "arguments": {
     "access_token": "token_travel_001",
     "card_id": "CC0000001",
     "travel_date": "2024-03-20",
     "travel_from": "SFO","travel_to": "JFK",
     "travel_class": "business"}}
  </tool_call>

[Tool Result]
  {"success": true,
   "data": {"booking_id": "BK0000017",
            "transaction_id": "TXN0000017"}}

[Assistant] Your business-class flight from
  SFO to JFK on March 20 is booked.
  Booking ID: BK0000017, charged $1,600
  to card CC0000001.
```

Validation. A separate validation agent evaluates each completed trajectory by replaying its tool calls against a fresh database copy. The verdict is a structured JSON object:

```
{"demand_fulfilled": true,
 "assistant_grounded": true,
 "user_consistent": true,
 "issues": [],
 "verdict": "accepted"}
```

When issues are detected, the verdict includes categorized findings:

```
{"demand_fulfilled": true,
 "assistant_grounded": false,
 "user_consistent": true,
 "issues": [{
   "type": "text_hallucination",
   "turn": 4, "message_index": 9,
   "detail": "The assistant claimed the
     flight includes lounge access, but
     no tool result mentions this benefit."
 }],
 "verdict": "rejected"}
```

B Training Data Preparation

The Fathoming experiment uses 500 environments produced by the initial Grafting run that preceded and motivated the scaled run reported in Section 4.4. Each environment undergoes one round of environment evolution (Section 3.1.4), after which we generate 30 tasks per environment using the task-generation procedure described in Section 3.1.3. Each task is simulated against a session-local database copy and validated by the trajectory-validation pipeline. Trajectories that pass automated verification are retained, yielding a pool of 12,745 verified multi-turn trajectories that serve as the shared trajectory pool for both Fathoming rounds.

C Training Configuration

Table 4 lists the GRPO training configuration used for both Fathoming rounds. For the reward in Section 3.2.3, the penalty terms are $P_{\text{extra}} = \min(0.20,\ 0.04\, n_{\text{extra}})$ for predicted calls that match no ground-truth call, $P_{\text{invalid}} = \min(0.20,\ 0.10\, n_{\text{malformed}})$ for syntactically unparseable `<tool_call>` blocks, and $P_{\text{dup}} = \min(0.15,\ 0.05\, n_{\text{dup}})$ for duplicate calls after canonicalization. Canonicalization sorts dictionary keys recursively, sorts list elements by JSON serialization, and normalizes strings by lowercasing, collapsing whitespace, and stripping.

Hyperparameter	Value
Base model	Qwen3-4B-Instruct-2507
Optimizer	AdamW
Learning rate	2×10^{-6}
Warmup steps	0
LR schedule	Constant
Batch size	128
GRPO group size (G)	16
KL penalty coeff.	0.001
Epochs	1
Max sequence length	30,720

Table 4: GRPO training configuration for Fathoming.

Bucket	Base	Round 1	Round 2
`too_easy`	62.7%	75.7%	76.5%
`effective`	5.0%	2.6%	3.5%
`hard`	26.4%	21.2%	19.9%
`too_hard`	5.8%	0.5%	0.1%

Table 5: Probing-bucket distribution across checkpoints on a 1,000-trajectory subset of the training data (7,567 sample-depth pairs).

D Capability Frontier Dynamics

To understand how the capability boundary evolves during Fathoming, we re-probe a random 1,000-trajectory subset of the 12,745-trajectory training pool against all three checkpoints: the base model, the round-1 checkpoint, and the round-2 checkpoint. Table 5 reports the four-bucket distribution over all 7,567 (sample, depth) pairs.

Three patterns emerge. First, the frontier is sparse: `effective` samples never exceed 5% of the pool, confirming that uniform random sampling would rarely encounter the capability-boundary signal that drives Fathoming's gains. Second, the frontier shifts substantially after training: tracking individual (sample, depth) pairs from the base model to the round-1 checkpoint reveals that 75% of initially `effective` pairs become `too_easy`, while 38% of initially `hard` pairs enter the `effective` or `too_easy` band. Without re-probing, the second round would train on stale selections that are predominantly too easy for the updated model. Third, format acquisition and semantic learning proceed at different rates: `too_hard` (the model cannot produce a valid tool-call format) drops from 5.8% to 0.1% across two rounds, whereas `hard` (correct format but wrong content) decreases only from 26.4% to 19.9%, indicating that the remaining bottleneck lies in predicting correct tool-call arguments rather than learning the output structure.

	Round 2	Round 2 + No-call	Δ
Overall	$39.7_{\pm 0.5}$	$37.7_{\pm 0.4}$	−2.0
Base	$56.8_{\pm 0.8}$	$51.1_{\pm 0.7}$	−5.7
Miss Func	$27.6_{\pm 1.4}$	$31.7_{\pm 1.1}$	+4.1
Miss Param	$27.9_{\pm 0.7}$	$27.1_{\pm 1.3}$	−0.8
Long Context	$46.5_{\pm 1.0}$	$40.7_{\pm 1.3}$	−5.8

Table 6: Effect of augmenting the round-2 training set with 300 no-call response targets (plain-text refusal and clarification turns rather than tool calls) on BFCLv4 multi-turn. Mean$_{\pm \text{std}}$ over five runs.

E No-Call Response Target Augmentation

To test whether no-call response targets can mitigate the Miss Function regression discussed in Section 4.1, we augment the round-2 training set with 300 trajectories whose ground-truth final turn is a plain-text response, either a clarifying follow-up question (when a required parameter is missing) or a polite refusal (when the requested capability is absent from the environment), rather than a tool call. The augmented model is trained from the same round-1 checkpoint and with the same GRPO configuration as the standard round-2 model; the only difference is the additional 300 refusal and clarification response targets.

Table 6 compares the two round-2 variants on BFCLv4 multi-turn.

The augmentation recovers most of the Miss Function regression (+4.1, from 27.6 to 31.7), confirming that the model can learn to decline tool calls when exposed to refusal targets. However, this comes at a substantial cost to Base (−5.7) and Long Context (−5.8): the small set of no-call response targets appears to suppress the model's willingness to attempt tool calls broadly, not only in scenarios where refusal is appropriate. Miss Param is essentially unchanged (−0.8), suggesting that learning to request clarification may require more targeted or abundant examples than the 300 mixed response targets used here. The net effect is a 2.0-point drop in overall accuracy, indicating that naively mixing a small number of no-call response targets into the frontier-selected training set does not resolve the positive-only signal limitation without incurring a *refusal tax* on the model's core tool-calling capability.

F Cost and Infrastructure

Token cost. We report token usage in two forms: measured per-unit averages and aggregate estimates for the two logged production compo-

Operation	Unit	Input	Output	Total
Env. synthesis	per skill	2.60M	37.2K	2.63M
accepted	per accepted skill	4.10M	59.8K	4.16M
rejected	per rejected skill	341K	3.3K	344K
Task gen.	per task	408K	5.9K	414K
Simulation	per task	150K	2.8K	153K
Validation/eval.	per task	856K	6.5K	862K
Fix/enrich	per environment	1.68M	18.9K	1.70M

Table 7: Average token usage for the synthesis and trajectory-production pipeline, split into input, output, and total tokens.

nents. Table 7 reports average token usage for the main pipeline operations, including environment synthesis, task generation, simulation, validation/evaluation, and fix/enrichment. The input column aggregates non-cached input, cache-read input, and cache-creation input; total tokens add output tokens. We avoid converting these values into dollar totals because provider pricing, caching policy, and routing configuration vary across runs.

Environment-synthesis total. Using the accepted-synthesis average in Table 7, constructing the final 5,101-environment pool corresponds to $5{,}101 \times 4.16\text{M} \approx 21.2\text{B}$ tokens.

Trajectory-data total. The trajectory corpus was produced over 500 training environments with 30 tasks per environment. Using the per-task averages in Table 7, task generation, simulation, and validation/evaluation contribute $500 \times 30 \times (414\text{K} + 153\text{K} + 862\text{K}) \approx 21.4\text{B}$ tokens. These aggregate values are extrapolated from measured averages and remain token estimates rather than provider-specific dollar costs.

Compute infrastructure. The main two-round Fathoming run was trained on a single server with 2 Intel Xeon Platinum 8480CL CPUs (112 cores / 224 threads), 1.5TB RAM, and 8 NVIDIA H20 GPUs (96GB memory each), using NVIDIA driver 550.144.03 on Ubuntu 22.04.5 LTS. Each probing round generates $K{=}8$ completions per (sample, depth) pair before GRPO training. The two GRPO rounds select 7,388 and 3,616 training samples, respectively. With batch size 128 and one epoch (Table 4), this corresponds to about 87 GRPO update steps. Each update step processes one global batch and takes approximately 30 minutes on all 8 GPUs, yielding about 44 wall-clock hours, or roughly 350 GPU-hours, for the main two-round training run. Additional ablations and failed exploratory runs are excluded from this compute accounting; API-token usage for environment and trajectory production is reported separately above.

G Environment Synthesis Entry Prompt and Skills

This appendix documents the prompt-level scaffolding used by the environment synthesis workers in Section 4.4. Each worker receives one raw skill at `raw_skills/<skill_id>/`, a sandboxed project directory, shared registry databases, and nine mounted synthesis skills. The runner assembles the entry prompt from the modules in Table 8; the mounted skills that the prompt points the agent to are listed in Table 9.

After the worker exits, the runner performs programmatic post-validation independent of the coding agent's self-report. It accepts a package only if no explicit rejection was written, `VIABILITY.json` does not reject the skill, all seven required files are present, and `uv run pytest` succeeds on `tests/test_runtime.py`. Successful packages are copied into the environment output directory and registered; failed or rejected skills are recorded in the registry with their reason and harness metadata.

H Task Synthesis Entry Prompt and Skills

This appendix documents the prompt-level scaffolding used by the task synthesis workers in Section 3.1.3. Each worker receives one packaged environment copied to `env/`, a shared task-pool database at `tasks.sqlite3`, an `output/` directory, a sampled task rule, optionally a sampled tool subgraph, and eight mounted task-generation skills. The runner assembles the entry prompt from the modules in Table 10; the mounted skills that the prompt asks the worker to execute are listed in Table 11.

After the worker exits, the runner performs programmatic post-validation independent of the coding agent's self-report. It accepts the task only if `output/task.json` exists, is valid JSON, uses a supported rule mode, has a non-empty `demand`, has list-valued `world_setup` SQL strings, and provides all three required `user_agent_config` strings. Successful tasks are copied into the task output directory and registered in `tasks.sqlite3` with harness provenance, token usage, rule mode, and the sampled subgraph fingerprint; failures are recorded with an error reason.

Entry module	Prompt content and purpose
Role	Defines the worker as an environment synthesis agent that either produces a complete executable environment package or rejects a non-viable skill.
Task	Points to `raw_skills/<skill_id>/` and instructs the worker to read `SKILL.md`, `scripts/`, `_meta.json`, and any other files to understand the target API, operations, and domain.
Mounted skills	Lists the nine harness-native skill directories under `.claude/skills/`, `.codex/skills/`, or `.opencode/skills/`, depending on the coding-agent backend. The prompt requires reading each `SKILL.md` before the corresponding synthesis step.
Resources	Exposes `skill_index.db`, `environment_index.db`, and `registry.db`. It gives command templates for searching related skills, checking duplicates, and checking whether a skill was already processed.
Workflow	Enforces an ordered lifecycle: read the raw skill; write `output/VIABILITY.json`; reject early via `output/REJECTED.json` if needed; check duplicates; write `tools.jsonl`, `schema.sql`, `world.db`, `tool_graph.json`, `backend.py`, `ENVIRONMENT.md`, and `test_runtime.py`; run quality validation; then run pytest and repair failures.
Constraints	States invariant package rules: all state in SQLite; a deterministic `EnvironmentBackend`; unified success/error envelopes; at least 12 tools; only `query` and `mutation` categories; no randomness, network, or external calls; TTL-based auth; masked sensitive data; CRUD completeness; sufficient seed data; conversational ID discoverability; and English-only generated artifacts.
Test environment	Specifies the validation context: `uv run pytest output/<env>/tests/test_runtime.py -x -q`, parent-directory working directory, 120-second timeout, stdlib-only backend imports, no subprocesses, no sockets, no external tool invocations, no environment-variable dependencies, and paths derived from `db_path`.
Output contract	Requires exactly the package files consumed downstream: `spec/tools.jsonl`, `spec/tool_graph.json`, `spec/ENVIRONMENT.md`, `runtime/backend.py`, `world/schema.sql`, `world/world.db`, and `tests/test_runtime.py`. A rejected skill must instead produce `output/REJECTED.json`.

Table 8: Environment synthesis entry-prompt modules assembled by the runner for each raw skill.

I Trajectory Simulation Prompt Templates

This appendix documents the prompt templates used when simulating user–assistant trajectories in Section 3.1.3. The simulation runner instantiates a role-playing user agent and a tool-using assistant agent, applies the task-specific `world_setup` SQL to a session-local copy of `world.db`, and dispatches assistant tool calls to the deterministic environment backend. Table 12 summarizes the prompt surfaces. The displayed templates are line-wrapped for readability.

I.1 Simulated User System Template

The user simulator's system prompt is parameterized by the generated task. The placeholders `{role}`, `{personality}`, and `{knowledge_boundary}` come from `task.user_agent_config`; `{demand}` comes from the task demand; and `{done_marker}` is `[DONE]`.

```
You simulate a real user in a multi-turn dialogue with a
customer service assistant.

## Your Persona
Role: {role}
Personality: {personality}
Knowledge boundary: {knowledge_boundary}

## Your Goal
{demand}

## Rules
- Stay fully in character. Use vocabulary and tone natural to
  your persona.
- Do NOT use tool names, API endpoints, parameter names, or
  system internals.
- Respond to the assistant's questions based on what your
  persona would know.
- If the assistant asks for information outside your knowledge
  boundary, say you don't know.
- When the assistant has fully completed your goal, output
  exactly: {done_marker}
- If the assistant cannot do what you asked and there is no way
  forward, output: {done_marker}
- Do NOT output {done_marker} until the goal is genuinely
  resolved or clearly impossible.
- Keep messages to 1-3 sentences. No markdown, no bullet points,
  no labels.

## Critical: Reveal Information Gradually
- In your FIRST message, only mention your primary reason for
  calling and give just enough context for the assistant to start
  helping (e.g. your name, that you need help with a booking).
  Do NOT list all your requests upfront.
- Reveal additional requests and specific details (dates, names,
  amounts) only when the assistant asks or after the current issue
  is resolved.
- A real person does not open a call by listing every single thing
  they need in one breath -- they start with the most important
  thing and bring up the rest as the conversation progresses.
```

I.2 Simulated User Turn Templates

At the start of a trajectory, the runner sends the user model the system template above plus the following instruction as the first `user` message:

```
The conversation is starting now. Generate your first message
to the assistant. Mention only your most immediate need -- do
not list everything upfront.
```

For subsequent turns, the runner appends the real assistant's latest natural-language response as a `user` message in the user model's conversation. The user model then generates the next simulated user message under the `assistant` role. The alternate single-prompt implementation uses this continuation instruction:

```
Generate your next message. Continue the conversation naturally
based on the agent's latest response.
```

I.3 Assistant System Template

The assistant simulator uses the following system prompt. During simulation, tools are supplied through the provider-native tool schema interface rather than printed directly into this prompt.

Mounted skill	Stage	Responsibility
skill-viability-assessment	Entry gate	Reads the raw skill, scripts, metadata, and related files; estimates extractable and expandable tool count; checks mockability and data constructability; applies hard rejection rules for external binaries, live credentials, real-time data, mixed runtimes, event-driven interfaces, file-backed state, and missing scripts; writes VIABILITY.json and optionally REJECTED.json.
tool-design	Tool surface	Extracts operations from scripts, expands the domain surface with meaningful CRUD, search, auth, relationship, aggregate, and state-transition tools, and writes spec/tools.jsonl with MCP-style descriptions, JSON schemas, enum constraints, query/mutation categories, unified return-format expectations, and conversational ID-discovery paths.
schema-design	World model	Designs world/schema.sql and populates world/world.db. It separates reference, mutable, session, transaction, junction, tree, and single-entity state tables; enforces naming and constraint conventions; and requires realistic, diverse English seed data with enough rows and edge cases for task generation.
tool-graph-design	Dependency graph	Builds spec/tool_graph.json. It creates one node per tool with category, read tables, and write tables, then adds data_dependency and state_precondition edges so downstream task generation can sample grounded multi-step workflows.
backend-implementation	Runtime	Implements runtime/backend.py as a stdlib-only Python EnvironmentBackend with _ok, _error, session/auth helpers, deterministic ID generation, a non-throwing call() dispatcher, mutation commits, masked sensitive data, and visible_state().
environment-brief	Specification brief	Writes spec/ENVIRONMENT.md after the other artifacts exist. The brief is grounded in the actual files and contains seven sections: title, overview, tools, key workflows, database schema, session initialization, and verification guidance.
runtime-tests	Self-tests	Writes tests/test_runtime.py. The required pytest suite dynamically loads the backend and covers smoke tests, at least one test per tool, multi-step workflows, error cases, protocol compliance, and database isolation.
quality-validation	Final quality gate	Reads all seven package files and produces a structured quality report. It checks tool-schema quality, tool graph and schema consistency, backend tool registration, ENVIRONMENT.md counts, test coverage, conversational accessibility, and seed-data volume and entropy.
synthesis-registry	Lifecycle utilities	Provides scripts for building and querying the raw-skill FTS index, detecting duplicate environments, finding related skills when a seed is too small, registering accepted packages, registering rejections, marking skills consumed, rebuilding the environment index, and reporting synthesis statistics.

Table 9: Mounted skills available to each environment synthesis worker. These are copied into the coding-agent-specific native skill directory before the worker is invoked.

```
You are an assistant operating in a tool-using multi-turn
conversation.

Your job is to help the user complete the current task by:
1. understanding the user's request,
2. deciding whether a tool is needed,
3. calling tools when they are actually useful,
4. giving the user a natural-language answer that is grounded in
   visible tool results.

Core rules:

- Be tool-first, not tool-only.
- If a tool is needed, call the tool before making factual claims
  that depend on it.
- If the available tools include discovery or listing tools
  (e.g. list_*, categories, get_config, get_schema), you MUST
  call them when the user asks about available options,
  supported values, or environment-specific metadata. Never
  answer such questions from your own knowledge alone.
- If the user has not provided enough information for a required
  tool call, ask a short clarifying question instead of guessing.
- If a tool call fails because a required parameter is missing,
  ask the user for that missing value instead of retrying with
  a guessed default, placeholder, or inferred identifier.
- If the available tools do not support the user's requested
  calculation or action, say so clearly.
- Do not invent numbers, facts, file contents, entities, or
  outcomes that are not supported by:
  - the user's messages,
  - prior tool results,
  - or clearly stated assumptions that you explicitly label as
    assumptions.
- Never present unsupported assumptions as computed results.
- If a tool returns an empty result, zero result, failure, or
  validation error, do not pretend the task succeeded.
- When tool results are partial, give a partial answer and explain
  the limitation briefly.

Tool-use policy:

- Use only available tools.
- Prefer the smallest set of tool calls that makes real progress.
- Do not call tools redundantly.
- Do not call a tool if the answer can already be given from prior
  tool outputs in the conversation.
- When multiple tools are relevant, use them in a sensible order.
- If no tool is needed for the current turn, respond directly in
  natural language.

Response policy:

- After tool use, provide a concise natural-language response to
  the user.
- Summarize the relevant result, not the raw tool protocol.
- Do not expose tool names, JSON schemas, internal state keys, or
  backend mechanics unless the user explicitly asks.
- Keep the answer focused on the user's current request.
- Do not jump ahead to unrelated future steps unless the user
  asks.
- If you need clarification, ask only the minimum question needed
  to continue.

Grounding policy:

- Any specific numeric result must be traceable to a tool result
  or explicit user-provided numbers.
- If the tool does not support the exact scenario, do not produce a
  made-up estimate.
- Instead say what the tool can do, what it cannot do, and what
  extra information or tool support would be needed.

Output policy:

- Normal case: return a helpful natural-language assistant
  message, and include tool calls when needed.
- Do not output hidden reasoning.
- Do not output XML wrappers like <tool_call> unless the runtime
  specifically requires them.
- Do not output raw JSON except when required for a tool call.

Priority order for each turn:

1. Stay grounded.
2. Make real progress with the available tools.
3. Ask for clarification only when necessary.
```

Entry module	Prompt content and purpose
Role	Defines the worker as a task generator that writes `task.json` files for multi-turn user–assistant trajectory simulation. It introduces the three core task components: a natural-language `demand`, SQL `world_setup`, and `user_agent_config`.
Rule	Injects one sampled rule mode and its instructions. The five modes are `normal`, `miss_func`, `miss_param`, `error_recovery`, and `long_context`; they respectively request standard achievable tasks, absent-capability tasks, underspecified-parameter tasks, recoverable-error tasks, and broader long-context tasks.
Target subgraph	Optional module emitted when `spec/tool_graph.json` is available. A programmatic sampler selects mutation anchors and predecessor query tools; the prompt states that the task must exercise the listed mutation tools and use the listed query tools as the discovery path, with data-flow edges shown explicitly.
Variation round	Optional high-volume module emitted when a rule–subgraph combination has already produced tasks. It requires a meaningfully different scenario by varying at least two of target entities, user persona, business scenario, and `world_setup`.
Environment resources	Lists the environment files available in the sandbox: `env/spec/ENVIRONMENT.md`, `env/spec/tools.jsonl`, `env/spec/tool_graph.json`, `env/world/schema.sql`, `env/world/world.db`, and `tasks.sqlite3` filtered by the current environment name.
Workflow	Requires the worker to read its mounted skills and then execute the ordered lifecycle: graph exploration, task creation, grounding check, solvability check, demand-clarity check, world-setup validation, task-pool conflict check, and mutation followed by re-validation if a conflict is found.
High-volume conflict guidance	Optional module for large target counts. It relaxes duplicate detection when the same mutation-tool set appears with different entities, personas, business scenarios, or initial database state, and flags conflicts only when both the mutation tools and core scenario are essentially identical.
Output contract	Requires the final artifact at `output/task.json` with exactly four fields: `demand`, `world_setup`, `user_agent_config` with `role`, `personality`, and `knowledge_boundary`, and `rule`. It explicitly forbids plural `demands`, deprecated `init_context`, and extra fields.

Table 10: Task synthesis entry-prompt modules assembled by the runner for each generated task.

Mounted skill	Stage	Responsibility
`tool-graph-exploration`	Coverage planning	Reads `tool_graph.json`, `tools.jsonl`, and `tasks.sqlite3`; identifies mutation anchors, prerequisite query tools, data-dependency edges, and existing coverage; and samples under-covered subgraphs adapted to the current rule.
`task-creation`	Draft construction	Creates a single complex `demand`, queries `world.db` for concrete entities, uses `world_setup` SQL to create task-specific state or preconditions, and writes a realistic `user_agent_config` with an explicit knowledge boundary.
`task-grounding-check`	Entity grounding	Verifies that every entity referenced in the demand exists in `world.db` or will exist after `world_setup` executes. Missing entities are fixed by changing the SQL setup or rewriting the demand.
`task-solvability-check`	Conversation reachability	Traces the tool chain from user-known information to required tool arguments. It rejects dead-locks where the user lacks an internal ID, credential, or lookup path needed to complete the task.
`task-demand-clarity-check`	Demand semantics	Checks that the demand has an unambiguous completion condition, avoids overlap, targets the intended mutations, and describes user goals rather than prescribed tool-call steps.
`world-setup-validation`	SQL validation	Runs each `world_setup` statement against a temporary copy of `world.db`, checks schema compatibility and post-setup state, and verifies authentication/session prerequisites when required.
`task-pool-conflict-check`	Duplicate filtering	Compares the candidate task against prior successful tasks for the same environment in `tasks.sqlite3`, using both mutation-tool overlap and semantic scenario similarity to detect near duplicates.
`task-mutation`	Repair and diversification	Rewrites failed or duplicate tasks while preserving the intended coverage when possible. It can change entities, reframe the scenario, adjust `world_setup`, modify the persona, or resample subgraphs, then reruns the validation skills.

Table 11: Mounted skills available to each task synthesis worker. They are copied into the coding-agent-specific native skill directory before the worker is invoked.

```
4. Keep the conversation natural and useful.
```

J Trajectory Validation Entry Prompt and Skills

This appendix documents the prompt-level scaffolding used by the trajectory validation workers in Section 3.1.3. Each worker receives one completed `trajectory.json`, a packaged environment copied to `env/`, an `output/` directory, and four mounted evaluation skills. The runner assembles the entry prompt from the modules in Table 13; the mounted skills referenced by the workflow are listed in Table 14.

After the worker exits, the runner performs programmatic post-validation independent of the coding agent's self-report. It requires `output/evaluation.json` to exist, parse as JSON, contain boolean values for `demand_fulfilled`, `assistant_grounded`, and `user_consistent`, contain a list-valued `issues` field, and use `accepted` or `rejected` as the verdict. The runner then recomputes the final verdict from the three booleans and the issue list; if the model-provided verdict disagrees, the recomputed verdict overrides it before the evaluation is copied to the output directory and registered in `evaluations.sqlite3`.

Prompt surface	Template artifact	Runtime use
Simulated user system prompt	user-simulator system template	The `_SYSTEM_TEMPLATE` is filled from each task's `demand` and `user_agent_config` fields: role, personality, and knowledge boundary. It controls the user's persona, goal, stopping rule, and gradual disclosure behavior.
Simulated user turn prompts	user-simulator turn template	The first call asks the user model to produce the opening user message. On later turns, the assistant's latest natural-language response is appended as a `user` message inside the user-model conversation, and the model's generated simulated-user reply is appended as an `assistant` message. This role inversion lets the user simulator exploit native chat turn-taking.
Assistant system prompt	assistant system template	The assistant receives a static system prompt that enforces grounded tool use, minimal clarification, and concise user-facing responses. Environment tools from `spec/tools.jsonl` are passed through the provider's native tool-schema channel during simulation.

Table 12: Prompt surfaces used for trajectory simulation.

Entry module	Prompt content and purpose
Role	Defines the worker as a trajectory evaluation agent for a simulated multi-turn user–assistant conversation. It asks the worker to decide whether the user's demand was fulfilled, whether the assistant stayed grounded in tool results and user-provided information, and whether the simulated user remained consistent with the task persona.
Task	Injects the task fields embedded in the trajectory: `demand`, optional `rule`, `world_setup` SQL, and `user_agent_config`. For `miss_func`, `error_recovery`, and `miss_param` tasks, the prompt adds rule-specific guidance so impossible sub-goals, recovery paths, and underspecified requests are validated under the intended task semantics.
Resources	Lists the files available in the sandbox: `trajectory.json`; `env/spec/ENVIRONMENT.md`; `env/spec/tools.jsonl`; `env/world/schema.sql`; `env/world/world.db`; and `env/runtime/backend.py`. These resources let the validator replay tool calls, inspect the environment schema, and verify final database state.
Workflow	Requires the worker to read its mounted skills and execute four checks in order: replay the recorded tool calls on a fresh environment copy; verify demand fulfillment against the post-replay database; check assistant text, operation claims, and tool-call arguments for hallucinations; and check user turns for knowledge-boundary violations, personality drift, AI-like language, and goal drift.
Output contract	Requires `output/evaluation.json` with exactly five top-level fields: `demand_fulfilled`, `assistant_grounded`, `user_consistent`, `issues`, and `verdict`. Each issue must include a typed category, 0-indexed turn, message index, and human-readable detail; accepted trajectories require all three booleans to be true and no issues.

Table 13: Trajectory validation entry-prompt modules assembled by the runner for each completed trajectory.

K Environment Evolution Entry Prompt and Skills

This appendix documents the prompt-level scaffolding used by the environment evolution workers in Section 3.1.4. Each worker receives one packaged environment copied to `env/`, assessment artifacts copied to `assess_artifacts/`, an `output/` directory, and five mounted evolution skills. The runner first performs a smoke assessment using task generation, simulation, and trajectory validation. It then assembles either a fix prompt or an enrichment prompt from the modules in Table 15; the mounted skills referenced by these prompts are listed in Table 16.

After the worker exits, the runner performs programmatic post-validation independent of the coding agent's self-report. The edited `env/` directory must still satisfy the packaged-environment structure check before it is copied back over the original environment, with the previous package saved as a backup. The runner then re-assesses the evolved environment using a fresh task-generation, simulation, and validation batch. If the reassessment yields zero accepted trajectories or any surviving `INTERNAL_ERROR`, the backup is restored and the environment is marked failed; otherwise the environment is marked evolved or enriched and retained for downstream trajectory production.

L Tool Diversity Scaling

Raw tool-name counts misrepresent diversity, since names like `search` recur across environments with different implementations. We embed each of the 130,985 tools with OpenAI's `text-embedding-3-small` (OpenAI, 2026), build a top-50 cosine-similarity graph via FAISS (Johnson et al., 2021), and count connected components, using τ=0.85 as the tightest threshold that merges near-duplicates without fusing opposites like `pause_trading`/`resume_trading` (merged only at τ=0.80); the full pool yields 83,350 *effective tools*. Under uniform random subsampling (Figure 6(a)), across $\tau \in \{0.80, 0.85, 0.90\}$ the count grows with log-log slope 0.94–0.98, and the per-env marginal at τ=0.85 stays at +15.6 even at the N=2,000 $\rightarrow$ 5,101 step. Meanwhile the Vendi score (Friedman and Dieng, 2023) saturates near 118 effective directions (Figure 6(b)), well below the 1,536-dimensional ambient ceiling: the pool keeps adding distinct functions at near-linear rate while their geometric spread saturates at a

Mounted skill	Stage	Responsibility
trajectory-replay	Deterministic replay	Runs scripts/replay_trajectory.py on trajectory.json and env/. The script copies env/world/world.db to output/eval.db, applies task world_setup, loads env/runtime/backend.py, replays every assistant tool call, compares replayed results with recorded tool results, and writes output/replay_result.json. It also substitutes replay-time auth tokens when deterministic state is otherwise preserved.
demand-verification	Goal verification	Decomposes the demand into concrete sub-goals and verifies each sub-goal against output/eval.db using SQL informed by env/world/schema.sql. It sets demand_fulfilled and records demand_not_fulfilled issues, while applying special handling for absent-capability, error-recovery, and missing-parameter task rules.
assistant-grounding	Assistant audit	Checks every assistant natural-language claim and function-call argument against the conversation context available before that message: previous user messages, previous tool results, and schema-grounded enum/default choices. It records text, operation, or parameter hallucinations when claims or arguments are not traceable, while allowing semantic paraphrases, logical consequences of successful operations, and authorized creative text fields.
user-consistency	User audit	Checks every simulated user message against user_agent_config. It flags knowledge-boundary violations, sustained personality drift, AI-like conversational style, and requests outside the original demand, while treating information revealed by the assistant earlier in the dialogue as newly available to the user.

Table 14: Mounted skills available to each trajectory validation worker. These are copied into the coding-agent-specific native skill directory before the worker is invoked.

Entry module	Prompt content and purpose
Role	Defines the worker as an environment evolution agent. In the fix variant, the role is to diagnose and repair quality issues found by a smoke assessment. In the enrich variant, the role is to add more complex seed data when all smoke-test trajectories were accepted and the environment appears too simple.
Assessment context	Injects the environment name, runner classification, classification reason, accepted/rejected counts, and the relevant sandbox paths. The prompt states that environment files are in env/ and assessment evidence is in assess_artifacts/.
Skills	Requires the worker to read the mounted skills before acting. Fix prompts execute assess-diagnosis, then backend-bugfix and/or contract-alignment, then runtime-verification. Enrichment prompts execute assess-diagnosis, seed-data-enrichment, and runtime-verification.
Resources	Lists the mutable environment package under env/, including spec/, runtime/, world/, and tests/; the copied assessment report, trajectories, and evaluations under assess_artifacts/; the harness-native skill directory under .claude/skills/, .codex/skills/, or .opencode/skills/; and the output/ directory for reports.
Constraints	States route-specific invariants. Fix prompts require minimal targeted changes, adherence to the environment contract, no changes outside the diagnosed issue scope, no third-party backend imports, and English-only seed data. Enrichment prompts restrict edits to env/world/world.db, forbid backend or tool-schema changes, require data-richness rules from the specification, preserve schema constraints and foreign keys, and require English-only seed data.
Output contract	Requires output/evolution_result.json summarizing the changes. The worker may also write intermediate structured reports such as diagnosis.json, fixes_applied.json, alignment_fixes.json, enrichment_report.json, and verification_result.json as specified by the mounted skills.

Table 15: Environment evolution entry-prompt modules assembled by the runner after the smoke assessment classifies an environment for fix or enrichment.

low-dimensional vocabulary intrinsic to natural-language tool descriptions.

Mounted skill	Stage	Responsibility
`assess-diagnosis`	Root-cause analysis	Reads `assess_artifacts/report.json`, trajectory files, evaluation files, `tools.jsonl`, `backend.py`, and `schema.sql`; classifies failures as environment bugs, contract mismatches, data issues, assistant limitations, or overly simple data; and writes `output/diagnosis.json` with recommended downstream skills.
`backend-bugfix`	Runtime repair	Applies targeted fixes to `env/runtime/backend.py` for diagnosed internal errors such as parameter mismatches, SQL table or column errors, missing handlers, foreign-key problems, and ID-prefix bugs. It treats `tools.jsonl` as authoritative for tool names and parameters and preserves the non-throwing `call()` interface.
`contract-alignment`	Artifact alignment	Repairs mismatches among `tools.jsonl`, `backend.py`, `tool_graph.json`, and `schema.sql`. It enforces consistency of tool names, parameter names, required/optional arguments, categories, and table references according to the authority order: tool definitions, schema, backend, then tool graph.
`seed-data-enrichment`	Data enrichment	Adds rows, diversity, edge cases, and balanced status distributions to `env/world/world.db`. It uses `schema.sql`, `tools.jsonl`, and `tool_graph.json` to respect foreign keys, CHECK constraints, ID formats, table roles, and data-richness requirements while leaving the schema and backend unchanged.
`runtime-verification`	Post-fix verification	Runs `scripts/verify_tool_calls.py` against `env/`. The script copies `world.db`, loads `EnvironmentBackend`, invokes each tool with minimal schema-valid arguments, treats `INTERNAL_ERROR` and `UNKNOWN_TOOL` as failures, allows ordinary business-logic errors such as authentication or validation failures, checks `visible_state()`, and writes `output/verification_result.json`.

Table 16: Mounted skills available to each environment evolution worker. These are copied into the coding-agent-specific native skill directory before the worker is invoked.

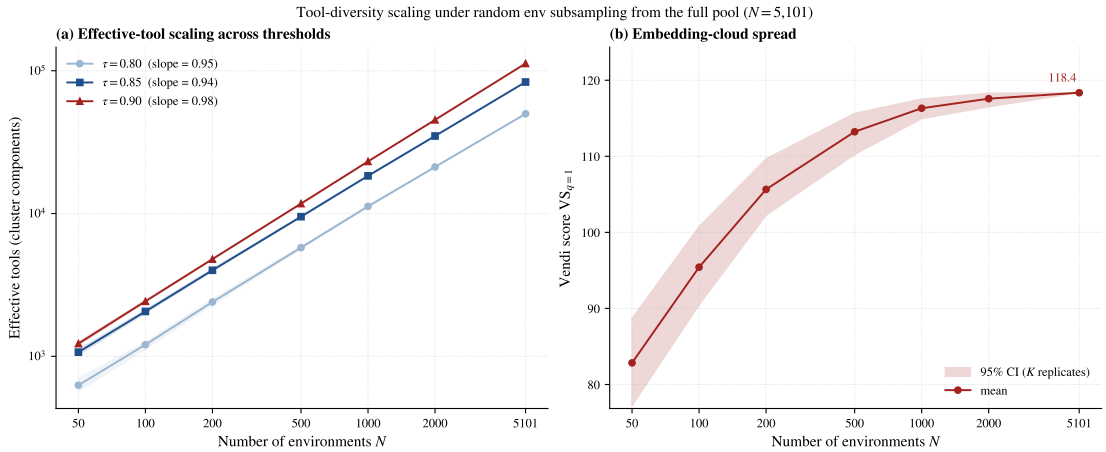

Figure 6: Tool diversity of the 5,101-environment pool under uniform random subsampling ($N \in \{50, 100, 200, 500, 1000, 2000, 5{,}101\}$). (a) Effective-tool count (connected components of the top-50 cosine-similarity graph over per-tool embeddings) versus pool size N at three thresholds; near-linear log-log slopes (0.94–0.98) indicate non-redundant growth. (b) Vendi score $\mathrm{VS}_{q=1}$ saturates near 118 effective directions, indicating new tools occupy directions already populated at smaller N. Shaded band: 95% CI across replicates.