Tool-R0: Self-Evolving LLM Agents for Tool-Learning from Zero Data

Anonymous ACL submission

Abstract

Large language models (LLMs) are becoming the foundation for autonomous agents that can use tools to solve complex tasks. Reinforcement learning (RL) has emerged as a common approach for injecting such agentic capabilities, but typically under tightly controlled training setups. It often depends on carefully curated task–solution pairs and substantial human supervision, which creates a fundamental obstacle to open-ended self-evolution. In this paper, we propose *Tool-R0* for training general purpose tool-calling agents from scratch with self-play RL, *under a zero-data assumption*. Initialized from the same base LLM, *Tool-R0* co-evolves a `Generator` and a `Solver` with complementary rewards: one proposes targeted challenging tasks at the other's competence frontier and the other learns to solve them with real-world tool calls. This creates a self-evolving cycle that requires no pre-existing datasets. Evaluation on five tool-use benchmarks show that *Tool-R0* yields **92.5%** relative improvement over the base model and surpasses fully supervised tool-calling baselines under the same setting. Our work further provides empirical insights into self-play LLM agents by analyzing co-evolution, curriculum dynamics, and scaling behavior.

Code **Web** **Models** **Logs**

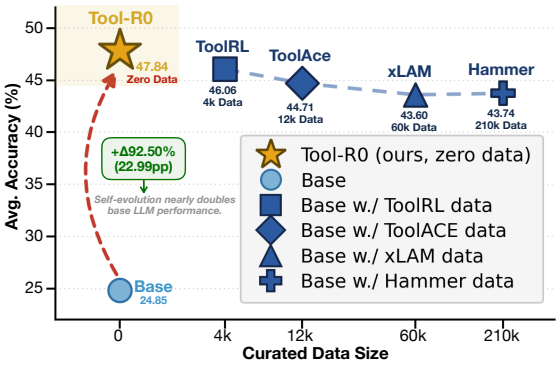

Figure 1: ***Tool-R0* self-evolves from zero curated data.** Self-play RL with a cross-verified, difficulty-aware curriculum lifts Qwen2.5-1.5B-Instruct from 24.85% to 47.84% average accuracy across five tool-calling benchmarks spanning heterogeneous APIs (**+22.99 pp**, +92.5%). *Tool-R0* matches or exceeds four supervised baselines re-trained on the *same* base model, with authors' released datasets and identical hyperparameters for fair comparison (see Appendix G for details).

1 Introduction

Large language models (LLMs) have demonstrated remarkable capabilities in complex reasoning tasks (Jaech et al., 2024; Guo et al., 2025). A key driver of this progress is reinforcement learning with verifiable rewards (RLVR), which has emerged as a powerful post-training paradigm for improving reasoning performance (Lambert et al., 2025). These advances position LLMs as promising language agents that can reason and act in real-world environments to accomplish goals by interacting with external tools (Yao et al., 2023; Qu et al., 2025). To teach LLMs with these agentic capabilities, large-scale RLVR has become the focal point (Zhang et al., 2025a; Qian et al., 2025).

Prior work on LLM agents has proposed different methods to improve their tool-calling skills over vast task datasets (Zhang et al., 2025c; Acikgoz et al., 2025a; Qian et al., 2025). However, curating such datasets through human annotation is inherently labor-intensive and fundamentally unscalable (Huang et al., 2023). As models advance, constructing large and high-quality datasets becomes increasingly unsustainable, creating a scalability bottleneck similar to that observed in LLM pre-training paradigm (Sutskever et al., 2024; Silver and Sutton, 2025). Consequently, reliance on human supervision poses a fundamental scalability bottleneck, limiting the ability of LLM agents to achieve self-evolving capabilities that a superintelligent AI would require (Zhang et al., 2025b; Gao et al., 2025). This naturally raises the following question: *Can we empower weak LLMs to improve themselves without requiring human data?*

Some of the most striking examples of superhuman AI emerge from self-play using RL in game environments (Silver et al., 2016, 2017; Bakhtin et al., 2022). Recently, interest in self-play has re-emerged with the success of LLMs trained via RL; where several works have applied it to reasoning tasks like math (Chen et al., 2024b; Cheng et al., 2024; Chen et al., 2025; Zhao et al., 2025; Liu et al., 2025a; Huang et al., 2025). Despite this progress, these methods do not extend self-play beyond abstract reasoning to tool-using agents in real-world environments. This further motivates whether self-play alone can incentivize LLMs with tool-calling skills across different domains under a zero-data assumption. Recent efforts such as Agent0 (Xia et al., 2025) and Dr. Zero (Yue et al., 2026) have begun to explore this direction. However, their focus remains narrow, either constrained to code tools for math problem-solving or search tools in question answering settings. In contrast, we focus on **how to train general-purpose tool-calling agents that can self-evolve base LLMs with zero external data across domains.**.

We propose *Tool-R0*, a self-play RL framework for training tool-calling agents that can self-evolve with zero human data (Fig. 1). In *Tool-R0*, a single base model is initialized with two different roles: a `Generator` and a `Solver` that are trained independently but co-evolve through complementary reward signals. The `Generator` is rewarded for generating challenging tasks aligned with the `Solver`'s evolving capabilities, while the `Solver` is trained to solve them with outcome-based rewards. To enable effective co-evolution, we design a difficulty-guided reward based on the frozen `Solver`'s answer uncertainty, prioritizing tasks at its competence frontier: **hard enough to teach and easy enough to verify.** We also follow a grounded task synthesis to ensure adaptive and controllable task generation, which conditions the `Generator` through annotation-free task configurations to prevent mode collapse in free-form generation. Since effective RL requires training data aligned with the model's current capabilities, we further order the constructed dataset by `Solver` answer consistency and progressively expose it to harder problems in each batch. Together, this self-evolving cycle significantly improves tool-calling performance over the base LLM and outperforms other supervised baselines across five different tool-calling benchmarks. We summarize the core insights derived from our study as follows:

- **Self-play can incentivize complex tool-calling skills from zero data.** Starting from weak priors, *Tool-R0* yields consistent gains across model scales, architectures, and benchmarks, showing that self-play RL alone is sufficient to teach complex tool-calling capabilities.
- **Adaptive self-generated curricula can surpass static human supervision.** Agents trained through self-play generalize better than those trained on fixed human-distribution datasets, demonstrating that LLMs can leverage themselves to identify and address their own capability gaps more effectively than fixed expert-designed curricula.
- **Role separation is necessary to avoid learning conflicts in self-play.** We observe that training `Generator` and `Solver` with separate parameters is essential, particularly when both roles operate over high-entropy action spaces with different reward objectives.
- **Co-evolution with difficulty-aware rewards drives learning.** We find that effective learning requires a dynamic `Generator` guided by smooth difficulty-aware rewards, since static generation or miscalibrated difficulty fails to sustain tool-use improvement.

Overview of contributions

- **(Algorithmic)** The primary contribution of this paper is *Tool-R0*, an adaptive self-play RL framework, where a `Generator` and a `Solver` co-evolve to build general-purpose tool agents with zero data from scratch.
- **(Performance)** *Tool-R0* consistently improves base models across architectures, scales, and benchmarks, and surpasses fully supervised tool agents, showing that LLMs can self-evolve into general tool-calling agents in a data-free setting.
- **(Analysis)** We conduct a series of experiments to analyze the key mechanisms of zero-data self-play for tool learning, including: role asymmetry, curriculum difficulty, adversarial co-evolution, self-play as mid-training stage, and scaling behaviors.
- **(Implementation)** We provide a modular and plug-and-play repository to support future research on self-evolving agents.

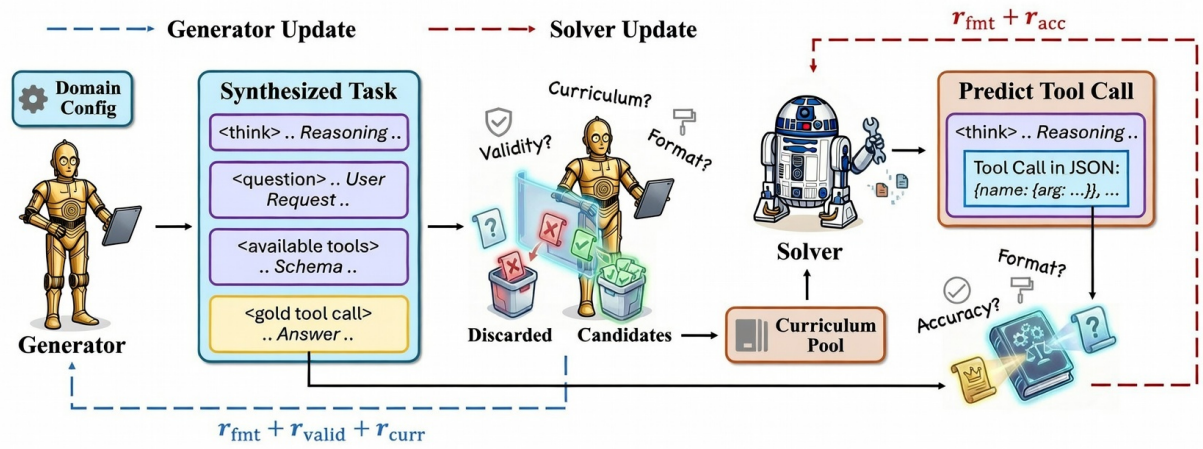

Figure 2: ***Tool-R0* Self-Evolution Framework.** A base LLM is initialized into two roles: a `Generator` and a `Solver`. The `Generator` synthesizes challenging tool-calling tasks (question, tool menu, and gold tool-call), targeting the frozen `Solver`'s competence frontier with designed rewards ($r_{\text{fmt}} + r_{\text{valid}} + r_{\text{curr}}$). Generated tasks are filtered and ranked easy-to-hard into a curriculum pool. The `Solver` trains on this curated data to predict tool calls ($r_{\text{fmt}} + r_{\text{acc}}$), completing a self-evolving cycle that requires no pre-existing human datasets.

2 Related Work

Tool-Learning with LLMs. Tool-learning has emerged as a promising paradigm for extending LLMs capabilities beyond parametric knowledge with external tools (Qu et al., 2025). It requires LLMs to perform structured reasoning over available tool schemas by selecting the right function, grounding arguments from context, and composing multi-step calls to accomplish goals (Yao et al., 2023; Schick et al., 2023; Patil et al., 2024). Several benchmarks have been proposed to systematically assess these capabilities across domains, task types, and tool sets (Tang et al., 2023; Srinivasan et al., 2023; Li et al., 2023; Wu et al., 2024). Building on this foundation, subsequent work has focused on constructing high-quality datasets and fine-tuning LLMs on such data to integrate more complex tool-calling skills (Zeng et al., 2024; Chen et al., 2024a; Zhang et al., 2025c; Liu et al., 2025c; Lin et al., 2025; Acikgoz et al., 2025a), including applying RLVR to further enhance tool-integrated reasoning (TIR) (Jin et al., 2025; Feng et al., 2025; Qian et al., 2025). However, their dependence on human supervision introduces limitations: (1) static human-curated data leads to distribution shifts that fail to capture the evolving needs of agents during on-policy learning and (2) limits the model's capabilities to human upper-bound.

Exploration of Self-Play with LLMs. Self-play has historically driven superhuman performance in closed-loop games (Tesauro, 1995; Silver et al., 2016; Bakhtin et al., 2022). Recently, interest in self-play has re-emerged with the success of LLMs trained via RLVR; where previous work extends self-play to open-domain settings like reasoning (Chen et al., 2024b; Cheng et al., 2024; Zhao et al., 2025; Fang et al., 2025; Chen et al., 2025; Liu et al., 2025a; Huang et al., 2025; Liu et al., 2025b). These works focus exclusively on abstract reasoning tasks like math, and do not address tool-using agents for real-world applications. More recent studies have begun exploring agentic self-play; Agent0 (Xia et al., 2025) applies self-play with Python coding tools and Dr. Zero (Yue et al., 2026) studies zero-data learning for search-centric question answering. Unlike prior self-play works in LLMs that focus on abstract reasoning or single-tool QA, *Tool-R0* targets *general-purpose* tool-calling across diverse domains with arbitrary API schemas.

3 Method

Overview. We propose *Tool-R0*, a zero-data self-play framework that co-evolves two complementary agents: a `Generator` that synthesizes verifiable tool-use tasks, and a `Solver` that learns to solve them—both initialized from the same base LLM (π). Training proceeds over K iterations, each comprising three stages: (1) the `Generator` (π_θ) is trained to produce grounded tool-calling tasks targeting the frozen `Solver`'s competence frontier (Sec. 3.2); (2) the trained `Generator` is frozen and construct a curated dataset via deduplication, verification, and curriculum ordering (Sec. 3.3); and (3) the `Solver` (π_ϕ) is trained on this dataset becomes the updated competence baseline (Sec. 3.4). As the `Solver` improves, the `Generator` must propose harder tasks to earn reward, creating a self-reinforcing cycle that emergent tool-use capabilities without any human data (full pseudocode in Appendix I).

3.1 Grounded Task Specification.

A key challenge in zero-data self-play is controllable and diverse task generations: when the **Generator** is prompted only with generic instructions to produce diverse tasks, the underlying language model concentrates probability mass on a small set of high-likelihood patterns, resulting in mode-collapse (Liu et al., 2025b). To prevent this, *Tool-R0* grounds task generation using lightweight task specifications $s = (d, c, m, n)$ where d denotes the task domain (Fig. 5), c the interaction context type, m the number of available tools, and n the number of gold tool-calls as answer. At each training step, specifications are dynamically sampled from a user-defined weighted distribution and injected into the **Generator**'s prompt (Fig. 10) as a form of meta-prompting, yielding a conditioned generation distribution $q \sim \pi_\theta(\cdot \mid s)$ that improves specificity, diversity, and robustness to hallucination. Formal definitions, sampling strategies, and hyperparameters are in Appendix B.

3.2 Generator Training

Tool-R0 first trains a **Generator** policy π_θ with Group Relative Policy Optimization (GRPO) (Shao et al., 2024) to synthesize challenging tool-calling tasks from scratch. The **Generator** receives only a prompt template (Fig. 10) and task specifications s. Each generated sample consists of: (i) a user request, (ii) an explicit tool menu, and (iii) a gold tool-call, within its private reasoning. To enforce structure, the **Generator** outputs four tagged blocks: `<think>`, `<question>`, `<available_tools>`, and `<tool_call_answer>`. The `<available_tools>` block defines the available tools as a JSON list, while `<tool_call_answer>` contains the corresponding gold tool calls as a JSON list with flat primitive arguments only. This design makes tool menus and calls are JSON-verifiable; enabling execution-based feedback rather than non-verifiable natural-language supervision (see Fig. 14 for a generated example).

3.2.1 Generator Reward Design

Training the **Generator** in a zero-data regime requires reward signals that simultaneously ensure output quality and learning utility. We therefore train the **Generator** with a set of rewards that serve three complementary purposes: (i) enforce a strict code verifiable output interface (r_{fmt}), (ii) guarantee internal consistency between the tool menu and the gold tool-call (r_{valid}), and (iii) induce an adaptive curriculum by targeting tasks that are neither trivial nor unsolvable for the current **Solver** (r_{curr}).

Format Reward (r_{fmt}): Tags and Parseability. Since downstream learning and verification require extracting the tool menu and gold tool-calls, we first enforce strict structural compliance. Let x denote a **Generator** completion and let $\mathbb{I}_{\text{tags}}(x)$ indicate that all required blocks can be extracted. We reward tag completeness and well-formed JSON artifacts:

$$r_{\text{fmt}}(x) = \mathbb{I}_{\text{tags}}(x) + \mathbb{I}_{\text{tools-json}}(x) + \mathbb{I}_{\text{gold-json}}(x), \tag{1}$$

where $\mathbb{I}_{\text{tools-json}}(x)$ indicates that `<available_tools>` parses as a JSON list of tool specs, and $\mathbb{I}_{\text{gold-json}}(x)$ indicates that `<tool_call_answer>` parses and normalizes into a canonical tool-call representation. This prevents reward hacking through malformed outputs.

Validity Reward (r_{valid}): Available Tools, Gold-Calls, and Value Grounding. Syntactic correctness alone does not guarantee that the proposed gold answer is realizable using the provided tools. We therefore enforce internal consistency between the tool menu, the gold tool-call, and the question to prevent action hallucination. Let $\mathcal{T}$ denote the parsed tool menu, $c^\star = (n^\star, a^\star)$ the normalized gold tool call with tool name $n^\star$ and flat argument map $a^\star$, and q is the generated question. We assign validity rewards based on three checks:

$$\begin{aligned} r_{\text{valid}}(x) = \lambda_{\text{Menu}}\, \mathbb{I}[n^\star \in \mathcal{T}] \\ + \lambda_{\text{Gold}}\, \mathbb{I}[\text{req}(n^\star) \subseteq \text{keys}(a^\star)] \\ + \lambda_{\text{Value}}\, \mathbb{I}[\text{vals}(a^\star) \hookrightarrow q]. \end{aligned} \tag{2}$$

where the three terms respectively verify that the gold tool exists in the menu, that all schema-required parameters are present, and that every non-trivial argument value (excluding booleans and nulls) appears as a word-boundary match in q ($\text{vals}(a^\star) \hookrightarrow q$). Prior self-play verifiers (Python execution in Absolute Zero and R-Zero; single-tool feedback in Agent0 and Dr. Zero) assume a fixed action space and are therefore not directly comparable. In heterogeneous tool-calling the proposer constructs that space itself, and r_{valid} is what uniquely makes its construction verifiable.

Curriculum Reward (r_{curr}): Difficulty & Semantic Alignment. Beyond syntactic validity, generated tasks must be *useful for learning*: they should (i) lie near the **Solver**'s current competence frontier and (ii) remain semantically faith-

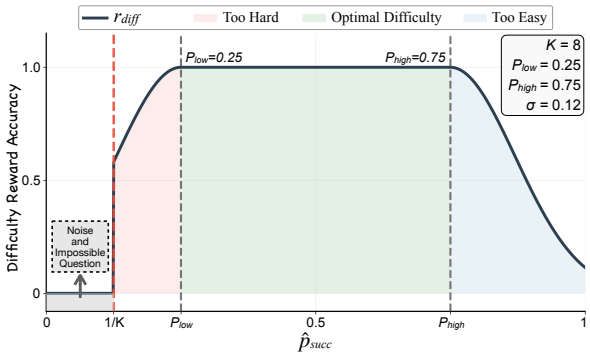

Figure 3: **Difficulty reward (r_{diff}).** A band-pass filter over the frozen **Solver**'s empirical success rate $\hat{p}_{\text{succ}}$: zero below $1/K$ to gate ill-posed tasks, plateau at 1 within $[P_{\text{low}}, P_{\text{high}}]$ to reward tasks at the competence frontier, and smooth Gaussian falloff outside to steer the **Generator** back toward learnable difficulty.

ful to the user request. We combine a **Solver**-calibrated difficulty signal (r_{diff}) with a semantic coherence signal (r_{sem}). Given a generated question and tool menu, we query the current **Solver** K times under stochastic decoding, obtaining predicted tool-calls $\{\hat{c}^{(k)}\}_{k=1}^{K}$, and estimate success as $\hat{p}_{\text{succ}} = \frac{1}{K}\sum_{k=1}^{K} \mathbb{I}[\hat{c}^{(k)} = c^{\star}]$. We assign maximal reward when $\hat{p}_{\text{succ}}$ falls within a target band $[P_{\text{low}}, P_{\text{high}}]$, with smooth Gaussian decay outside:

$$r_{\text{diff}}(x) = \begin{cases} 0, & \hat{p}_s < 1/K, \\ \exp\left(-\frac{(\hat{p}_s - P_{\text{low}})^2}{2\sigma^2}\right), & \hat{p}_s < P_{\text{low}}, \\ 1, & \hat{p}_s \in [P_{\text{low}}, P_{\text{high}}], \\ \exp\left(-\frac{(\hat{p}_s - P_{\text{high}})^2}{2\sigma^2}\right), & \hat{p}_s > P_{\text{high}}. \end{cases} \quad (3)$$

This reward acts as a **band-pass filter** (Fig. 3), encoding three design principles. **(1) Gate unsolvable tasks.** When no **Solver** sample matches the gold call ($\hat{p}_{\text{succ}} < 1/K$), the reward is zero; where generations are likely ill-posed and provide no learning signal. **(2) Reward a plateau, not a peak.** Rather than targeting a single probability, we reward an interval centered near $\hat{p}_{\text{succ}} \approx 0.5$, ensuring tasks are neither trivial nor intractable while absorbing the noise inherent to finite-sample Monte Carlo estimation. **(3) Smooth falloff.** The Gaussian tails outside the plateau provide a continuous gradient: too-hard tasks receive diminishing reward that steers the **Generator** toward easier variants, and too-easy tasks toward harder one, without destabilizing RL training.

3.3 Solver Dataset Construction

After training, we freeze the **Generator** and use it as a task synthesizer. We first sample a large pool of candidate tasks and remove near-duplicates via canonicalized signatures derived from question–tool–call combinations. To increase the reliability of pseudo-labels, we cross-verify each candidate by sampling multiple predictions from **Solver** and measuring agreement with the generated gold tool-call by retaining only tasks with consistent solutions; following the principle that reproducible answers provide more reliable supervision (Huang et al., 2023; Zuo et al., 2025). From this verified pool, we estimate difficulty via pass@K success rates and group tasks into easy, medium, and hard buckets. The final dataset is selected to preserve domain diversity while maintaining a balanced difficulty mix, organized as a curriculum progressing from easier to harder instances (see Appendix D).

3.4 Solver Training

Solver (π_ϕ) is trained to predict correct tools given a user query q and tool menu $\mathcal{T}$ using a prompt template that elicits explicit TIR (Fig. 12), following prior work (Guo et al., 2025). It first generates reasoning within `<think>` tags, followed by predicted tool calls in `<tool_call_answer>` tags. We use two different rewards for our training: (i) a format reward r_{fmt} and (ii) a dense accuracy reward r_{acc}.

Format Reward (r_{fmt}) and Accuracy Reward (r_{acc}). To stabilize training and avoid unstructured outputs, we assign partial credit on parseability for our special tokens:

$$r_{fmt}(\hat{y}) = \lambda_{tag} \cdot \mathbb{I}_{\text{tag}} + \lambda_{parse} \cdot \mathbb{I}_{\text{parse}} + \lambda_{norm} \cdot \mathbb{I}_{\text{norm}}, \quad (4)$$

where $\hat{y}$ is the model completion and normalization maps common tool-call variants into the canonical schema. For parsed outputs, we decompose tool-call correctness into three components following prior work (Qian et al., 2025). Given predicted tool-calls $\hat{C}$ and gold tool-calls $C^{\star} = [c_1^{\star}, \ldots, c_n^{\star}]$, we match each gold tool-call to the best unused prediction (greedy maximum match). For a matched pair $(\hat{c}, c^{\star})$, we compute three dense sub-scores: (i) name match $s_{\text{name}} \in \{0, 1\}$, (ii) key overlap $s_{\text{key}} \in [0, 1]$ measured by F1 over argument keys, and (iii) value match $s_{\text{val}} \in [0, 1]$ defined as the fraction of matching values over the intersection of keys, using a robust comparator (numeric coercion and whitespace-insensitive string matching). The pair score is

$$s(\hat{c}, c^{\star}) = \lambda_{\text{name}} s_{\text{name}} + \lambda_{\text{key}} s_{\text{key}} + \lambda_{\text{val}} s_{\text{val}} \quad (5)$$

4 Experiments

Experimental Settings. We use Qwen2.5-1.5B-Instruct (Yang et al., 2025) as our primary model,

Name	ToolAlpaca	SealTool	NexusRaven	API-Bank	SNIPS	Avg
Qwen Models						
Qwen2.5-0.5B-Instruct	20.17	37.07	4.71	13.85	1.57	15.47
w/ *Tool-R0*	31.58	63.95	17.61	28.00	14.29	30.57
Δ	+11.41	+26.88	+12.90	+14.15	+12.72	+15.62
	↑ 56.57%	↑ 72.51%	↑ 273.89%	↑ 102.17%	↑ 810.19%	↑ 101.03%
Qwen2.5-1.5B-Instruct	35.96	47.27	17.61	19.13	4.29	24.85
w/ *Tool-R0*	47.36	83.00	34.59	50.62	20.86	47.84
Δ	+11.40	+35.73	+16.98	+31.49	+16.57	+22.99
	↑ 31.70%	↑ 75.59%	↑ 86.42%	↑ 164.61%	↑ 386.25%	↑ 92.52%
Qwen2.5-3B-Instruct	45.61	69.72	44.33	44.95	14.28	43.97
w/ *Tool-R0*	53.51	78.23	47.8	47.94	15.57	48.50
Δ	+7.90	+8.51	+3.47	+2.99	+1.29	+4.53
	↑ 17.32%	↑ 12.21%	↑ 7.83%	↑ 6.65%	↑ 9.03%	↑ 10.30%
Llama Models						
Llama-3.2-3B-Instruct	35.96	68.70	45.60	27.08	12.29	36.12
w/ *Tool-R0*	43.86	77.21	46.86	30.24	14.42	40.47
Δ	+7.90	+8.51	+1.26	+3.16	+2.13	+4.35
	↑ 21.97%	↑ 12.39%	↑ 2.76%	↑ 11.67%	↑ 17.33%	↑ 12.04%

Table 1: **Main results of *Tool-R0*.** We evaluate five benchmarks targeting distinct tool-use skills. Highlighted rows denote models trained with *Tool-R0*. +Δ reports absolute accuracy gain over, percentages show relative improvement.

with 0.5B and 3B variants for scaling analysis and Llama-3.2-3B-Instruct (Grattafiori et al., 2024) for cross-family generalization. We deliberately adopt pre-Qwen3 backbones to minimize *benchmark contamination*, enable *matched-backbone re-training* of all supervised baselines under identical hyperparameters (Table 2), and isolate the contribution of self-play from base-model capability. We run three self-play iterations of 50 steps; per iteration, the `Generator` is trained on 2,000 self-generated samples and then frozen to synthesize 10,000 candidate tasks. A full run takes ∼4 hours on 4× A40 GPUs. We evaluate on five tool-calling benchmarks: Toolalpaca (Tang et al., 2023), Seal-Tools (Wu et al., 2024), NexusRaven (Srinivasan et al., 2023), API-Bank (Li et al., 2023), and SNIPS (Coucke et al., 2018) (See Appendix F for details).

4.1 Results

Research Question 1: Can *Tool-R0* enable LLMs to learn complex tool-calling skills through self-play from scratch? We show the main results of *Tool-R0* in Table 1. For our primary model, Qwen2.5-1.5B-Instruct, we observe an average improvement of +22.99pp (92.52% relative gain). These gains span diverse settings, including single-turn API selection (SealTool), multi-step tool composition (ToolAlpaca, NexusRaven), conversational tool use (API-Bank), and user-intent tracking (SNIPS), indicating *strong task generalization* rather than overfitting to specific tool distributions. These results affirm that self-play RL with *Tool-R0* enables weak base LLMs to self-evolve into general-purpose tool-calling agent with substantial gains across all evaluated benchmarks.

Research Question 2: Does *Tool-R0* generalize across model scales and families? *Tool-R0* yields consistent gains across both axes (Table 1). *Across scales*, it narrows the capability gap between smaller and larger models: the 0.5B model reaches 30.57% (surpassing the 1.5B base) and the 1.5B model reaches 47.84% (surpassing the 3B base), eliciting latent tool-use even from sub-billion-parameter models, though absolute gains diminish as base capability grows (+15.62 at 0.5B → +4.53 at 3B). *Across families*, Qwen2.5-3B and Llama-3.2-3B both improve under identical hyperparameters (+4.53; +4.35), indicating that *Tool-R0* is **architecture-agnostic** and that gains track *initial base capability* rather than model family.

Takeaway 1

Tool-R0 enables LLMs to self-evolve into general-purpose tool-calling agents entirely from scratch without any human data, yielding consistent gains across different benchmarks, model scales, and architectures.

Model	#data	ToolAlpaca	SealTool	Nexus-Raven	API-Bank	SNIPS	Avg
Agents Trained on Existing Curated Tool-Calling Data							
Qwen2.5-1.5B-Instruct	–	35.96	47.27	17.61	19.13	4.29	24.85
⊢w/ xLAM Dataset (Zhang et al., 2025c)	60k	51.75	69.05	38.68	34.65	**23.85**	43.60
⊢w/ Hammer Dataset (Lin et al., 2025)	210k	45.61	68.70	**51.88**	33.10	19.42	43.74
⊢w/ ToolAce Dataset (Liu et al., 2025c)	12k	45.61	67.01	43.08	53.71	14.14	44.71
⊢w/ ToolRL Dataset (Qian et al., 2025)	4k	46.49	72.78	34.14	**62.04**	14.86	46.06
***Tool-R0* Training w/ No Curated Data (Ours)**							
Tool-R0 (Ours)	**0**	**47.36**$^{+11.40}$	**83.00**$^{+35.73}$	34.59$^{+16.98}$	50.62$^{+31.49}$	20.86$^{+16.57}$	**47.84**$^{+22.99}$

Note: *For fair comparison, all models are trained using Qwen2.5-1.5B-Instruct with identical hyperparameters on their respective public datasets; see Appendix G.*

Table 2: **Comparison with Supervised Baselines.** All baselines re-trained on the identical Qwen2.5-1.5B-Instruct backbone with matched hyperparameters and authors' released datasets. + marks absolute gain over base model.

Research Question 4: How does *Tool-R0* compare to other supervised models trained with human expert data? Table 2 compares *Tool-R0* against models supervised fine-tuned on existing curated tool-calling datasets, where all models share the same Qwen2.5-1.5B-Instruct backbone (see Appendix G for details). *Tool-R0* achieves 47.84% average accuracy with *zero* curated data, outperforming methods trained on 4k-210k human-annotated examples. We note that these baselines are themselves not purely human-annotated: their training corpora are synthesized by strong external LLMs (GPT-4 class) with downstream filtering, so Table 2 effectively compares *Tool-R0* against four variants of LLM-supervised tool-calling, making the gap a comparison of self-supervision against external-model supervision rather than against curation alone. The strongest baseline, ToolRL, excels on specific benchmarks but averages only 46.06%; suggesting overfitting to distributional patterns in curated datasets. These results highlight the surprising effectiveness of our approach that pretrained LLMs can generate automated curricula superior to human-designed ones by directly addressing capability gaps the model itself identifies, showing that *the model itself knows best what data it needs*.

Figure 4: Self-play coverage analysis.

4.1 Why does self-play outperform curated supervision? We compute pairwise cosine similarity between each training corpus and the five test benchmarks (Fig. 4). Curated datasets show *narrow and uneven* benchmark coverage, reflecting the static biases of their original supervision, whereas *Tool-R0*'s self-generated curriculum achieves the highest average similarity *and* the most uniform coverage across all five benchmarks despite never observing test data. The *uniformity*, not just the magnitude, of this coverage suggests adaptivity to the learner's evolving capability gaps rather than drift toward any particular benchmark.

Takeaway 2

The key advantage of self-play is adaptivity: unlike human-curated datasets, which encode a fixed supervision, self-play continually reshapes training toward the model's current weaknesses, yielding broader coverage and stronger generalization.

4.2 Ablation Studies and Analysis

Research Question 5: How does parameter sharing affect `Generator`-`Solver` co-evolution? Table 3 shows that, despite all variants improving over the base model, sharing parameters causes a substantial performance drop (-17.42 pp). We attribute this to two complementary factors. First, tool calling requires inherently asymmetric roles: the `Generator` must explore an unbounded task distribution, while the `Solver` must reliably execute precise actions under fixed API semantics. Symmetric role-play with shared weights works well in closed, single-objective domains such as Go or Poker, but not in this open-ended, high-entropy action space. Second, heterogeneous `Generator` and `Solver` rewards induce gradient interference under shared parameters: exploration-driven gradients from the `Generator` conflict with execution-driven gradients from the `Solver`, producing unstable representations that neither role retains. Real-world asymmetric agentic tasks require parameter separation; without it, co-evolution collapses.

Takeaway 3

Asymmetric `Generator`-`Solver` objectives make parameter separation necessary, as shared weights induce gradient interference and unstable co-evolution.

Method	Avg. Acc.	Δ Acc. Drop	
		pp	Relative (%)
Tool-R0	47.84	–	–
Ablations			
⊢ shared weights	30.42	-17.42	↓36.41%
⊢ frozen `Generator` (❄)	41.65	-6.19	↓12.94%
⊢ w/o Difficulty Reward (Eq. (3))	43.54	-4.30	↓8.99%
⊢ w/o Gaussian falloff (Eq. (3))	44.10	-3.74	↓7.82%

Table 3: **Ablation analysis relative to *Tool-R0*.** Absolute changes are reported in percentage points (pp), and relative changes as percentage drop (↓%).

Research Question 6: Does the `Generator` meaningfully improve through self-play? We triangulate three independent signals (ablation, training dynamics, and qualitative evolution), all pointing to the same conclusion: *Tool-R0* depends on a *learning* `Generator`, not a static prompt-induced sampler. **(i)** Freezing the `Generator` after initialization drops average accuracy by -6.19 pp (Table 3), showing that gains require its task distribution to track the `Solver`'s evolving competence. **(ii)** Reward dynamics (Fig. 6, Bottom Right) show the difficulty component rising from 0.1 to 0.4 across Iterations 1–2 *while semantic coherence remains stable*: harder tasks, not noisier ones. **(iii)** Qualitative examples (Fig. 13 and 14) show clear progression from single-tool requests to compositional, constraint-heavy, multi-step problems.

Research Question 7: Does difficulty calibration matter, or is solvable generation enough? The band-pass difficulty signal r_{diff} (Eq. (3)) steers the `Generator` toward the `Solver`'s competence frontier. Removing it reduces accuracy by 4.30 points (↓8.99% relative; Table 3), showing that generating *solvable* tasks is insufficient: the `Generator` must also calibrate task difficulty. Training dynamics (Fig. 6, Bottom Right) corroborate this: the difficulty component rises steadily across iterations while semantic coherence remains stable, indicating that the `Generator` targets the desired challenge level without sacrificing task validity.

7.1 Role of smooth Gaussian transitions. Our band-pass reward uses Gaussian decay outside $[p_{\text{low}}, p_{\text{high}}]$ rather than hard cutoffs. Replacing the Gaussian tails with a rectangular filter (reward 1 inside, 0 outside) degrades accuracy (Table 3, last row). Hard cliffs eliminate gradient signal near the band edges, whereas smooth transitions remain *informative*: their slope is proportional to difficulty drift and steers the `Generator` back toward the competence frontier even when generations fall outside the target band.

Takeaway 4

Self-play succeeds only when the `Generator` tracks the `Solver`'s competence frontier via a calibrated, band-pass curriculum: static or uncalibrated generation collapses the learning signal.

Additional Analysis. Appendix H extends *Tool-R0* along five axes: (i) *Tool-R0* as a *mid-training stage* that amplifies subsequent supervised post-training beyond either approach alone; (ii) qualitative evidence that the `Generator` progressively shifts from single-tool requests to compositional, multi-step problems; (iii) reward-dynamics analysis of `Generator`-`Solver` co-evolution, with parallels to adversarial training stability; (iv) an honest characterization of self-play *saturation* across model scales, identifying when and why improvement plateaus; and (v) a fine-grained breakdown of tool-use failure modes, isolating where self-play resolves errors and where it does not. Together, these analyses show that *Tool-R0*'s gains arise from *measurable curriculum adaptation* rather than reward gaming, and delineate the regimes where the framework is most and least effective.

5 Conclusion

We introduced *Tool-R0*, a self-play RL framework that trains tool-calling models over *heterogeneous APIs* using **zero curated tool-calling data**. Across five benchmarks and four model scales, a Generator-Solver loop with verifiable rewards consistently matches or surpasses supervised baselines re-trained on the same backbone with 4k–210k curated examples, suggesting that an adaptive, self-generated curriculum can substitute for static human-curated supervision in this regime. We attribute this to a structural property: curated datasets encode a fixed snapshot of expert priors, whereas self-play continuously targets the model's evolving capability gaps via measurable, difficulty-calibrated signals rather than self-reported reward. These findings suggest that for tasks with verifiable structure, self-play is a viable alternative to human-curated supervision and a useful mid-training stage that amplifies it.

Limitations

Tool-R0 is an early step, and we believe its most exciting implications lie ahead. While *Tool-R0* currently operates over general tool-calling task, extending self-play to environments like long-horizon web agents, embodied environments, and multi-turn conversations is an important next step. Beyond that, the most critical component of any self-play system is the signal that tells the **Generator** what the **Solver** currently needs, i.e., learnability reward, which we formalize as the difficulty reward in Eq. (3). We used **Solver** consistency as a proxy, but more interpretable alternatives, such as gradient-based learnability measures, semantic error attribution, or uncertainty estimation, could yield more informative curriculum signals. Finally, our results suggest that larger models saturate later and retain more headroom under self-play, hinting at favorable scaling behavior. A systematic study of scaling laws of self-play would provide the community with principled guidance for building self-evolving agents, which we leave as exciting future work.

Ethics Statement

This paper presents *Tool-R0* whose goal is to advance self-evolving agent research. The framework lowers the human supervision, and data annotation burden typical of supervised tool-learning pipelines, and the resulting models are studied in controlled, schema-verifiable settings. We see no specific negative social impacts. All experiments use publicly available models and benchmarks consistent with their respective licenses. We release code, models, and training logs under the MIT license to support reproducibility and unrestricted future research.

References

Emre Can Acikgoz, Jeremiah Greer, Akul Datta, Ze Yang, William Zeng, Oussama Elachqar, Emmanouil Koukoumidis, Dilek Hakkani-Tür, and Gokhan Tur. 2025a. Can a single model master both multi-turn conversations and tool use? CoALM: A unified conversational agentic language model. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12370–12390, Vienna, Austria. Association for Computational Linguistics.

Emre Can Acikgoz, Cheng Qian, Heng Ji, Dilek Hakkani-Tür, and Gokhan Tur. 2025b. Self-improving llm agents at test-time. *arXiv preprint arXiv:2510.07841*.

Anton Bakhtin, Noam Brown, Emily Dinan, Gabriele Farina, Colin Flaherty, Daniel Fried, Andrew Goff, Jonathan Gray, Hengyuan Hu, Athul Paul Jacob, Mo jtaba Komeili, Karthik Konath, Minae Kwon, Adam Lerer, Mike Lewis, Alexander H. Miller, Sandra Mitts, Adithya Renduchintala, Stephen Roller, and 7 others. 2022. Human-level play in the game of diplomacy by combining language models with strategic reasoning. *Science*, 378:1067 – 1074.

Jiaqi Chen, Bang Zhang, Ruotian Ma, Peisong Wang, Xiaodan Liang, Zhaopeng Tu, Xiaolong Li, and Kwan-Yee K. Wong. 2025. SPC: Evolving self-play critic via adversarial games for LLM reasoning. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Zehui Chen, Kuikun Liu, Qiuchen Wang, Wenwei Zhang, Jiangning Liu, Dahua Lin, Kai Chen, and Feng Zhao. 2024a. Agent-FLAN: Designing data and methods of effective agent tuning for large language models. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 9354–9366, Bangkok, Thailand. Association for Computational Linguistics.

Zixiang Chen, Yihe Deng, Huizhuo Yuan, Kaixuan Ji, and Quanquan Gu. 2024b. Self-play fine-tuning converts weak language models to strong language models. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 6621–6642. PMLR.

Pengyu Cheng, Tianhao Hu, Han Xu, Zhisong Zhang, Yong Dai, Lei Han, nan du, and Xiaolong Li. 2024. Self-playing adversarial language game enhances LLM reasoning. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Alice Coucke, Alaa Saade, Adrien Ball, Théodore Bluche, Alexandre Caulier, David Leroy, Clément Doumouro, Thibault Gisselbrecht, Francesco Caltagirone, Thibaut Lavril, and 1 others. 2018. Snips voice platform: an embedded spoken language understanding system for private-by-design voice interfaces. *arXiv preprint arXiv:1805.10190*.

Wenkai Fang, Shunyu Liu, Yang Zhou, Kongcheng Zhang, Tongya Zheng, Kaixuan Chen, Mingli Song, and Dacheng Tao. 2025. SeRL: Self-play reinforcement learning for large language models with limited data. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

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. *arXiv preprint arXiv:2504.11536*.

Huan-ang Gao, Jiayi Geng, Wenyue Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, Shilong Liu, Jiahao

Qiu, Xuan Qi, Yiran Wu, and 1 others. 2025. A survey of self-evolving agents: On path to artificial super intelligence. *arXiv preprint arXiv:2507.21046*.

Ian J Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. *Advances in neural information processing systems*, 27.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Chengsong Huang, Wenhao Yu, Xiaoyang Wang, Hongming Zhang, Zongxia Li, Ruosen Li, Jiaxin Huang, Haitao Mi, and Dong Yu. 2025. R-zero: Self-evolving reasoning llm from zero data. *arXiv preprint arXiv:2508.05004*.

Jiaxin Huang, Shixiang Gu, Le Hou, Yuexin Wu, Xuezhi Wang, Hongkun Yu, and Jiawei Han. 2023. Large language models can self-improve. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 1051–1068, Singapore. Association for Computational Linguistics.

Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, and 1 others. 2024. Openai o1 system card. *arXiv preprint arXiv:2412.16720*.

Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan O Arik, Dong Wang, Hamed Zamani, and Jiawei Han. 2025. Search-r1: Training LLMs to reason and leverage search engines with reinforcement learning. In *Second Conference on Language Modeling*.

Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James Validad Miranda, Alisa Liu, Nouha Dziri, Xinxi Lyu, Yuling Gu, Saumya Malik, Victoria Graf, Jena D. Hwang, Jiangjiang Yang, Ronan Le Bras, Oyvind Tafjord, Christopher Wilhelm, Luca Soldaini, and 4 others. 2025. Tulu 3: Pushing frontiers in open language model post-training. In *Second Conference on Language Modeling*.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023. API-bank: A comprehensive benchmark for tool-augmented LLMs. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 3102–3116, Singapore. Association for Computational Linguistics.

Qiqiang Lin, Muning Wen, Qiuying Peng, Guanyu Nie, Junwei Liao, Jun Wang, Xiaoyun Mo, Jiamu Zhou, Cheng Cheng, Yin Zhao, Jun Wang, and Weinan Zhang. 2025. Robust function-calling for on-device language model via function masking. In *The Thirteenth International Conference on Learning Representations*.

Bo Liu, Leon Guertler, Simon Yu, Zichen Liu, Penghui Qi, Daniel Balcells, Mickel Liu, Cheston Tan, Weiyan Shi, Min Lin, and 1 others. 2025a. Spiral: Self-play on zero-sum games incentivizes reasoning via multi-agent multi-turn reinforcement learning. *arXiv preprint arXiv:2506.24119*.

Bo Liu, Chuanyang Jin, Seungone Kim, Weizhe Yuan, Wenting Zhao, Ilia Kulikov, Xian Li, Sainbayar Sukhbaatar, Jack Lanchantin, and Jason Weston. 2025b. Spice: Self-play in corpus environments improves reasoning. *arXiv preprint arXiv:2510.24684*.

Weiwen Liu, Xu Huang, Xingshan Zeng, xinlong hao, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Zhengying Liu, Yuanqing Yu, Zezhong WANG, Yuxian Wang, Wu Ning, Yutai Hou, Bin Wang, Chuhan Wu, Wang Xinzhi, Yong Liu, Yasheng Wang, and 8 others. 2025c. ToolACE: Winning the points of LLM function calling. In *The Thirteenth International Conference on Learning Representations*.

Zichen Liu, Changyu Chen, Wenjun Li, Penghui Qi, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. 2025d. Understanding r1-zero-like training: A critical perspective. In *Second Conference on Language Modeling*.

Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*.

Arindam Mitra, Luciano Del Corro, Guoqing Zheng, Shweti Mahajan, Dany Rouhana, Andres Codas, Yadong Lu, Wei-ge Chen, Olga Vrousgos, Corby Rosset, and 1 others. 2024. Agentinstruct: Toward generative teaching with agentic flows. *arXiv preprint arXiv:2407.03502*.

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2024. Gorilla: Large language model connected with massive APIs. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

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. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

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.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Advances in neural information processing systems*, 36:68539–68551.

Jürgen Schmidhuber. 2011. Powerplay: Training an increasingly general problem solver by continually searching for the simplest still unsolvable problem. *Frontiers in Psychology*, 4.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, and 1 others. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, L. Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy P. Lillicrap, Fan Hui, L. Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. 2017. Mastering the game of go without human knowledge. *Nature*, 550:354–359.

David Silver and Richard S Sutton. 2025. Welcome to the era of experience.

Venkat Krishna Srinivasan, Zhen Dong, Banghua Zhu, Brian Yu, Hanzi Mao, Damon Mosk-Aoyama, Kurt Keutzer, Jiantao Jiao, and Jian Zhang. 2023. Nexusraven: A commercially-permissive language model for function calling. In *NeurIPS 2023 Foundation Models for Decision Making Workshop*.

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2024. Neurips 2024 test of time award session: Sequence to sequence learning with neural networks. Conference session.

Qiaoyu Tang, Ziliang Deng, Hongyu Lin, Xianpei Han, Qiao Liang, Boxi Cao, and Le Sun. 2023. Toolalpaca: Generalized tool learning for language models with 3000 simulated cases. *ArXiv*, abs/2306.05301.

Gerald Tesauro. 1995. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3):58–68.

Leandro von Werra, Younes Belkada, Lewis Tunstall, Edward Beeching, Tristan Thrush, Nathan Lambert, Shengyi Huang, Kashif Rasul, and Quentin Gallouédec. 2020. TRL: Transformers Reinforcement Learning.

Yinjie Wang, Ling Yang, Ye Tian, Ke Shen, and Mengdi Wang. 2025a. Co-evolving llm coder and unit tester via reinforcement learning. *arXiv preprint arXiv:2506.03136*.

Zengzhi Wang, Fan Zhou, Xuefeng Li, and Pengfei Liu. 2025b. Octothinker: Mid-training incentivizes reinforcement learning scaling. In *2nd AI for Math Workshop @ ICML 2025*.

Mengsong Wu, Tong Zhu, Han Han, Chuanyuan Tan, Xiang Zhang, and Wenliang Chen. 2024. Seal-tools: Self-instruct tool learning dataset for agent tuning and detailed benchmark. In *CCF International Conference on Natural Language Processing and Chinese Computing*, pages 372–384. Springer.

Peng Xia, Kaide Zeng, Jiaqi Liu, Can Qin, Fang Wu, Yiyang Zhou, Caiming Xiong, and Huaxiu Yao. 2025. Agent0: Unleashing self-evolving agents from zero data via tool-integrated reasoning. *arXiv preprint arXiv:2511.16043*.

Fangzhi Xu, Hang Yan, Chang Ma, Haiteng Zhao, Qiushi Sun, Kanzhi Cheng, Junxian He, Jun Liu, and Zhiyong Wu. 2025. Genius: A generalizable and purely unsupervised self-training framework for advanced reasoning. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13153–13167, Vienna, Austria. Association for Computational Linguistics.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*.

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. *arXiv preprint arXiv:2601.07055*.

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

Guibin Zhang, Hejia Geng, Xiaohang Yu, Zhenfei Yin, Zaibin Zhang, Zelin Tan, Heng Zhou, Zhongzhi Li,

Xiangyuan Xue, Yijiang Li, and 1 others. 2025a. The landscape of agentic reinforcement learning for llms: A survey. *arXiv preprint arXiv:2509.02547*.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. 2025b. Darwin godel machine: Open-ended evolution of self-improving agents. *arXiv preprint arXiv:2505.22954*.

Jianguo Zhang, Tian Lan, Rithesh R N, Zhiwei Liu, Weiran Yao, Juntao Tan, Yihao Feng, Thai Quoc Hoang, Tulika Manoj Awalgaonkar, Liangwei Yang, Shelby Heinecke, Huan Wang, Juan Carlos Niebles, Silvio Savarese, and Caiming Xiong. 2024. The agent ohana: Designing unified data and training pipeline for effective agent learning. In *ICLR 2024 Workshop on Large Language Model (LLM) Agents*.

Jianguo Zhang, Tian Lan, Ming Zhu, Zuxin Liu, Thai Hoang, Shirley Kokane, Weiran Yao, Juntao Tan, Zhiwei Liu, Yihao Feng, Juan Carlos Niebles, Shelby Heinecke, Huan Wang, Silvio Savarese, and Caiming Xiong. 2025c. xLAM: A family of large action models to empower AI agent systems. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 11583–11597, Albuquerque, New Mexico. Association for Computational Linguistics.

Andrew Zhao, Yiran Wu, Yang Yue, Tong Wu, Quentin Xu, Yang Yue, Matthieu Lin, Shenzhi Wang, Qingyun Wu, Zilong Zheng, and Gao Huang. 2025. Absolute zero: Reinforced self-play reasoning with zero data. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. 2024. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, Bangkok, Thailand. Association for Computational Linguistics.

Yifei Zhou, Sergey Levine, Jason E Weston, Xian Li, and Sainbayar Sukhbaatar. 2025. Self-challenging language model agents. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Yuxin Zuo, Kaiyan Zhang, Li Sheng, Shang Qu, Ganqu Cui, Xuekai Zhu, Haozhan Li, Yuchen Zhang, Xinwei Long, Ermo Hua, Biqing Qi, Youbang Sun, Zhiyuan Ma, Lifan Yuan, Ning Ding, and Bowen Zhou. 2025. TTRL: Test-time reinforcement learning. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Appendix

A Related Work Extended

Tool Learning with LLMs. Tool-integrated reasoning (TIR) enables LLMs to ground parametric knowledge through external tools such as APIs, databases, and software functions (Qu et al., 2025). Early work primarily focused on benchmarking and evaluation, measuring tool selection, argument generation, and execution correctness across curated datasets (Li et al., 2023; Tang et al., 2023; Srinivasan et al., 2023; Wu et al., 2024; Patil et al., 2024). Building on these benchmarks, subsequent efforts emphasized data construction and supervised fine-tuning, producing large-scale, human- or model-generated instruction corpora to teach tool use (Mitra et al., 2024; Chen et al., 2024a; Zhang et al., 2024). More recent systems combine such data with stronger post-training pipelines to improve robustness and generalization (Zhang et al., 2025c; Liu et al., 2025c; Lin et al., 2025; Acikgoz et al., 2025a), while reinforcement learning has been applied as an auxiliary stage to refine function-calling accuracy (Qian et al., 2025).

Despite steady progress, all existing approaches fundamentally rely on curated supervision—either explicit demonstrations, synthetic instruction data, or static task distributions. This reliance limits scalability to new domains where realistic user requests, correct tool traces, and verifiable outcomes are unavailable. In contrast, we study tool learning under a strict zero-data assumption, where no demonstrations, prompts, or external task corpora are accessible, and show that tool-use skills can emerge purely through self-play and autonomous curriculum generation.

Self-Play in LLMs. Self-play has long driven advances in AI for superintelligence, from early curiosity-driven two-agent setups (Schmidhuber, 2011) to game mastery in TD-Gammon (Tesauro, 1995), AlphaGo (Silver et al., 2016, 2017), and CICERO (Bakhtin et al., 2022). In LLMs, self-play initially focused on alignment via methods like SPIN (Chen et al., 2024b) evolving to capability enhancement in verifiable domains such as code generation with Coder-Tester pairs (Wang et al., 2025a) or adaptive problem creation from zero data (Zhao et al., 2025; Fang et al., 2025). Recent efforts like SPAG (Cheng et al., 2024), SPC (Chen et al., 2025), Genius (Xu et al., 2025), and SPIRAL (Liu et al., 2025a) use games or seeded tasks for reasoning gains, yet pure self-play often stalls—R-Zero shows marginal gains and degrades after iterations (Huang et al., 2025), Absolute Zero enhances performance in coding and mathematics but limited to coding-bound as verifiable environment (Zhao et al., 2025), and Agent0 relies on single-tool Python calls (Xia et al., 2025). More importantly, these methods are restricted to general knowledge or math environments and remain largely underexplored for agentic tasks that require complex, real-world tool use. While Zhou et al. (2025) proposes Code-as-Task generation for targeted and more challenging training data, it is limited by code-based verification (similar to Absoulte Zero (Zhao et al., 2025)), depends on structured tools and environments rather than a true zero-data paradigm, and optimizes only the `Generator`, precluding genuine `Generator`–`Solver` co-evolution. Crucially, none of these approaches address general-purpose tool-using agents operating over heterogeneous, real-world APIs, where actions are high-entropy, verification is execution-based, and task distributions must evolve alongside model competence. A direct numerical comparison against these methods is structurally ill-posed: their verifiers (Python execution in Absolute Zero, answer-equivalence in R-Zero, single-tool feedback in Agent0 and Dr. Zero) do not produce a reward signal over heterogeneous JSON tool-call outputs, and substituting our tool-calling verifier into their pipelines would yield a new method rather than a baseline. Moreover, several recent works optimize only the task generator as challenger (Zhou et al., 2025), precluding genuine co-evolution and often leading to unstable or saturated learning dynamics.

Our work fills this gap by introducing a dual-agent self-play framework in which a task-generating `Generator` and an executing `Solver` co-evolve symbiotically under complementary reward signals. **Unlike prior self-play methods, While concurrent self-play methods (Absolute Zero, R-Zero, Agent0, Dr. Zero) establish important precedents, none transfers directly to general-purpose tool-calling: their verifiers (Python execution, math-answer equivalence, single-tool search) presuppose a fixed action space, whereas tool-calling requires the proposer to generate the action space itself (a heterogeneous JSON tool menu) for which a new class of verifier (Eq. (2)) is required. Designing this verifier and the surrounding task-specification machinery is the central technical problem we address.**

Finally, *Tool-R0* targets tool-integrated reasoning across diverse domains, operates in a fully zero-data regime, and demonstrates sustained capability gains driven by adaptive, self-generated curricula rather than static task generation.

B Grounded Task Specification

B.1 Formal Definitions

To ensure domain-agnostic yet controllable task generation, the `Generator` is conditioned on an explicit domain configuration specification that constrains both the semantic scope and structural properties of each generated task. See Fig. 5 for the standardized domain configuration templates used in our experimental setup. This design is conceptually related to the grounding strategy of Liu et al. (2025b), where constrained generation improves specificity, diversity, and robustness to hallucination. However, unlike Liu et al. (2025b), which grounds generation in large, static documents, our approach relies on lightweight, dynamically adjustable configurations that encode user-level preferences. This enables rapid adaptation across users and domains while preserving control, making grounding a flexible interface rather than a fixed knowledge source. In our setting, each training example is associated with a sampled specification

$$s = (d, c, m, n) \tag{6}$$

where d denotes the task domain, c the interaction context type, m the number of available tools, and n the number of gold tool calls.

B.2 Domain Configurations

The `Generator` receives this specification through a structured system prompt (See Fig. 10) that enforces strict adherence to the desired domain and interface. The prompt requires the model to (i) generate a natural user question grounded in the *specified domain*, (ii) define an *explicit tool menu of size* m with JSON-verifiable schemas, and (iii) provide exactly n *number of gold tool calls* with flat primitive arguments.[1] This design allows us to control generation without providing any task-level supervision, while ensuring that every generated instance admits automated verification and execution-based feedback. Specifications $s = (d, c, m, n)$ are sampled independently for each training example. The sampling strategy is intentionally *non-uniform*, reflecting realistic tool-use distributions and emphasizing domains where precise tool selection and argument grounding are critical.

[1]To prevent ungrounded or templated outputs, the prompt explicitly forbids meta-instructions, placeholders, or abstract descriptions, and requires all argument values in the gold tool calls to appear verbatim in the user question.

B.3 Sampling Strategy and Hyperparameters

Domain sampling. Domains are drawn from a fixed weighted uniform distribution over more than 30 functional and agentic categories, including finance, healthcare, scheduling, databases, cloud infrastructure, and system utilities as in Fig. 5. Crucially, this **distribution is entirely user-defined**: practitioners specify both the set of domains and their relative weights, allowing the `Generator`'s data curriculum to directly reflect task importance, deployment priorities, or safety constraints. Higher weights may be assigned to precision-critical domains (e.g., finance, healthcare, productivity), while exploratory or open-ended domains (e.g., web search, social media) can be intentionally down-weighted. We treat these values as unnormalized sampling preferences rather than probabilities, enabling flexible reconfiguration without retraining or architectural changes.

Hyperparameters for Task Specifications. The interaction context is sampled as: (i) single-turn with probability 0.9 and (i) multi-turn with probability 0.1. Multi-turn prompts embed short conversational histories inside the user question and are restricted to a single gold tool call to reduce ambiguity. For single-turn contexts, the number of gold calls is sampled as $n = 1$ with probability 0.8 and $n = 2$ with probability 0.2. For multi-turn contexts, n is fixed to 1. When $n > 1$, the tool menu size is sampled from $\{3, 4, 5\}$ to limit combinatorial ambiguity. When $n = 1$, the menu size is sampled from two buckets: small menus (2–4 tools) or larger menus (5–8 tools), mimicking evaluation benchmarks with varying tool density. This specification sampling scheme encourages compositional diversity while maintaining solvability under automated verification.

C Generator Implementation Details

This section specifies the hyperparameters and implementation details used to instantiate the `Generator` training procedure described in Sec. 3.2. Table 5 summarizes all reward-related hyperparameters used in `Generator` training. Unless otherwise stated, all values are fixed across

```
DOMAIN_WEIGHTS = {
    "finance": 0.03125, "healthcare": 0.03125,
    "productivity": 0.03125, "retail_ecommerce": 0.03125,
    "scheduling": 0.03125, "database": 0.03125,
    "cloud_infrastructure": 0.03125, "system": 0.03125,
    "programming": 0.03125, "geolocation": 0.03125,
    "logistics": 0.03125, "communication": 0.03125,
    "iot": 0.03125, "cybersecurity": 0.03125, "insurance": 0.03125,
    "legal": 0.03125, "news": 0.03125, "weather": 0.03125,
    "sports": 0.03125, "entertainment": 0.03125,
    "education": 0.03125, "real_estate": 0.03125,
    "food_ordering": 0.03125, "translation": 0.03125,
    "utilities": 0.03125, "government": 0.03125,
    "memory_management": 0.03125, "web_search": 0.03125,
    "social_media": 0.03125, "math": 0.03125,
    "vehicle_control": 0.03125, "travel": 0.03125
}
```

Figure 5: **Domain sampling configuration.** The Generator samples task domains from a fixed set of functional and agentic categories using a uniform prior. Weights are treated as unnormalized sampling coefficients rather than probabilities, currently uniform.

experiments.

C.1 Training Setup

Table 4 summarizes the **Generator**'s training hyperparameters. Our **Generator** is trained with Grouped Relative Policy Optimization (GRPO) (Shao et al., 2024; Guo et al., 2025) using the TRL (von Werra et al., 2020) implementation. Training is performed with HuggingFace Accelerate and DeepSpeed ZeRO-3 (Rajbhandari et al., 2020) in mixed precision (`bfloat16`) on three GPUs. We use a per-device batch size of 2 with gradient accumulation over 4 steps, resulting in a global batch size of 24 sequences per update. The optimizer is AdamW (Loshchilov and Hutter, 2017) with a fixed learning rate of 1×10^{-6}. We train for 50 steps per run and we sample 4 generations per prompt during GRPO updates to compute relative advantages. The maximum completion length is set to 4096 tokens, which accommodates full task specifications including tool menus and gold tool calls.

C.2 Reward Hyperparameters

Structured output interface. Each **Generator** completion is required to emit exactly four tagged blocks: `<think>`, `<question>`, `<available_tools>`, and `<tool_call_answer>`, as defined in §3.2. The `<available_tools>` block must parse as a JSON list of tool specifications, and the `<tool_call_answer>` block must parse as a JSON list of tool calls with flat primitive arguments only. This constraint enables deterministic parsing and execution-based reward computation.

Hyperparameter	Generator	Solver
Dataset Size	2000	2000
Global Batch Size	24	32
Optimizer	AdamW	AdamW
Learning Rate	1×10^{-6}	1×10^{-6}
Weight Decay	1×10^{-2}	1×10^{-2}
KL Penalty Coefficient (λ_{KL})	1×10^{-2}	1×10^{-2}
Max Steps	50	50
Number of Group Rollouts	4	4
Rollout Temperature	1.0	1.0
Precision	BFloat16	BFloat16
Max Sequence Length	4096	4096

Table 4: **Training hyperparameters.** Details of the main training configurations for the **Generator** and **Solver** during Self-Play iterations.

Reward composition. The **Generator** is trained with three reward components: format reward r_{fmt} (Eq. (1)), validity reward r_{valid} (Eq. (2)), and curriculum reward r_{curr}.

The format reward is a sum of three binary indicators as defined in Eq. (1): tag completeness ($\mathbb{I}_{\text{tags}}$), tool-menu JSON validity ($\mathbb{I}_{\text{tools-json}}$), and gold-call JSON validity ($\mathbb{I}_{\text{gold-json}}$).

The validity reward assigns weights $(\lambda_{\text{menu}}, \lambda_{\text{gold}}, \lambda_{\text{value}}) = (0.4, 0.4, 0.2)$ to three checks: (i) the gold tool name exists in the parsed menu, (ii) all schema-required parameters are present in the gold call, and (iii) every non-trivial argument value (excluding booleans and nulls) appears as a word-boundary match in the generated question ($\text{vals}(a^\star) \hookrightarrow q$), as described in Eq. (2). This acts as a compiler-like gate: the first two checks prevent hallucinated calls to non-existent

tools or calls with missing arguments, while the third acts as a semantic anchoring constraint that ties the gold answer back to the generated question, discouraging solutions whose arguments have no evidential basis in the task. When <tool_call_answer> contains a list of calls, we canonicalize all calls into a normalized tool-call representation for verification.

The curriculum reward $r_{\text{curr}} = r_{\text{diff}} + r_{\text{sem}}$ is an unweighted sum of the difficulty and semantic alignment components.

Difficulty estimation. As described in Eq. (3), solver-calibrated difficulty is estimated using Monte Carlo sampling. We query the current **Solver** $K = 8$ times per task with temperature 0.7 and maximum generation length 2048 tokens. Note that this temperature (0.7) is used exclusively for difficulty estimation during **Generator** reward computation, and is distinct from the **Solver**'s training rollout temperature (1.0) listed in Table 4. Tasks with empirical success probability $\hat{p}_{\text{succ}} < 1/K$ (i.e., no solver sample matches the gold tool-call) receive zero difficulty reward, as they likely represent ill-posed, ambiguous, or unsolvable generations that provide no meaningful learning signal. For solvable tasks, difficulty is shaped using the band-pass function in Eq. (3) with parameters $P_{\text{low}} = 0.25$, $P_{\text{high}} = 0.75$, and $\sigma = 0.12$.

Semantic alignment. Semantic alignment between the generated user question and the gold tool call is evaluated using the **Solver** as a judge. The **Solver** assigns an integer score in $\{1, \ldots, 5\}$ for semantic coherence (Fig. 11), which is normalized to $[0, 1]$ as $r_{\text{sem}} = (s - 1)/4$. This signal penalizes vague or templated questions even when the corresponding tool call is syntactically valid.

Normalization. Gold tool calls are normalized into a canonical (`name`, `arguments`) representation prior to verification. Schema validation enforces presence of required parameters but does not perform strict type checking to keep reward computation inexpensive.

D Details of Solver Dataset Construction

After training the **Generator**, we freeze it and use it purely as a task synthesizer conditioned on the same control specifications described in Sec. 3.1. Starting from these domain-conditioned inputs, we construct **Solver** training data through a three-stage pipeline consisting of *generation and deduplication*, *cross-verification*, and *difficulty probing and selection*.

Hyperparameter	Value
Validity Reward (Eq. (2))	
λ_{menu} (tool existence)	0.4
λ_{gold} (required arguments)	0.4
λ_{value} (value grounding)	0.2
Curriculum Reward (Eq. (3))	
Solver samples (K)	8
Solver temperature (difficulty estimation)	0.7
Solver max tokens (difficulty estimation)	2048
Difficulty band ($P_{\text{low}}, P_{\text{high}}$)	(0.25, 0.75)
Gaussian width σ	0.12
Semantic alignment scale	$\{1, \ldots, 5\} \rightarrow [0, 1]$

Table 5: **Generator reward hyperparameters.** Hyperparameter values for the validity and curriculum reward components used during **Generator** training (Sec. 3.2). The format reward (Eq. (1)) uses unweighted binary indicators and has no tunable parameters.

D.1 Specification-Grounded Generation and Deduplication

We first sample a large pool of 10,000 candidate tasks from the frozen **Generator**, producing structured triples of user requests, tool menus, and gold tool calls. To avoid training bias caused by repeated patterns, we remove near-duplicate samples via canonicalized signatures derived from question–tool–call combinations, producing a large but non-redundant candidate pool.

D.2 Solver-Based Cross-Verification

To further increase the reliability of generated pseudo-labels, we sample each candidate multiple times using the **Solver** and measure agreement between predicted and generated gold tool calls, retaining only tasks with consistent solutions and discarding instances with low agreement. This procedure follows the principle that reproducible answers provide more reliable supervision, as consistently reproduced solutions are more likely to correspond to correct supervision signals (Huang et al., 2023; Zuo et al., 2025; Acikgoz et al., 2025b). Consequently, this stage filters ambiguous or noisy pseudo-labels and retains only tasks whose solutions provide reliable signal.

D.3 Difficulty Probing and Curriculum Selection

From the verified pool, we estimate task difficulty via **Solver** pass@K success rates and group the generated tasks into easy, medium, and hard buckets. Samples are then selected to preserve domain

diversity while maintaining a balanced difficulty mix, preventing bias toward trivially solvable tasks. Through this pipeline, the initial 10,000 candidates are filtered down to 2,000 samples that form the final **Solver** training set for each self-play iteration.

The selected data are organized into a staged curriculum progressing from easier to harder instances at the training batch level: early batches are composed primarily of easy samples, while harder tasks are progressively introduced in later batches. Data difficulty is critical for effective RL, as training data must align with the model's current capabilities to avoid learning failure (Liu et al., 2025d; Wang et al., 2025b). When the **Solver** is exposed to tasks far beyond its current competence too early, policy gradients become noisy and uninformative, leading to unstable optimization or degenerate solutions. By categorizing synthesized tasks by **Solver** answer consistency and progressively exposing the **Solver** to harder problems across batches, we ensure that the model builds foundational tool-calling skills on reliable examples before tackling compositional multi-tool scenarios that require those skills as prerequisites.

This design ensures that the final dataset is simultaneously diverse, semantically valid, and appropriately challenging, while avoiding noisy or degenerate pseudo-labels that could destabilize **Solver** training. In contrast to static synthetic pipelines, our procedure continuously grounds generation quality in **Solver** behavior, yielding a self-consistent data distribution suitable for stable agentic learning.

E Solver Implementation Details

This section lists the hyperparameters and implementation details used to instantiate **Solver** training in Sec. 3.4. Similar to **Generator**, we train the **Solver** with GRPO using two verifiable rewards: a graded format reward r_{fmt} and a dense accuracy reward r_{acc}. However, different from **Generator** our **Solver** prompt follows a Tool-Integrated Reasoning (TIR) interface: the model emits reasoning in `<think>` and a predicted tool-call list in `<tool_call_answer>`, as described in Sec. 3.4.

E.1 Training Setup

Training is performed under the same infrastructure as the **Generator**: HuggingFace Accelerate with DeepSpeed ZeRO-3 (Rajbhandari et al., 2020) in mixed precision (`bfloat16`) on three GPUs. We use a per-device batch size of 2 with gradient accumulation over 5 steps, yielding a slightly larger global batch size of 32 sequences per update compared to the **Generator**'s 24. This increase reflects the higher variance in **Solver** rollouts due to the open-ended nature of tool-call prediction, where larger batches help stabilize advantage estimation across diverse task difficulties. The optimizer is AdamW (Loshchilov and Hutter, 2017) with a fixed learning rate of 1×10^{-6} and weight decay of 1×10^{-2}, matching the **Generator** configuration. We train for 50 steps per self-play iteration on the curated dataset of 2,000 samples constructed through the pipeline described in Sec. 3.3, with tasks ordered from easy to hard based on the **Solver**'s own answer consistency. During GRPO updates, we sample 4 generations per prompt at temperature 1.0 to compute relative advantages. The maximum completion length is set to 4096 tokens, which accommodates the full TIR output including extended reasoning traces and multi-call tool predictions. The KL penalty coefficient is set to $\lambda_{\text{KL}} = 1 \times 10^{-2}$ to regularize the policy against the reference model. All hyperparameters are summarized alongside the **Generator** configuration in Table 4.

E.2 Reward Hyperparameters

Output interface and parsing. A completion is considered structurally valid if it contains a non-empty `<tool_call_answer>` block. We parse `<tool_call_answer>` with a super-relaxed loader that accepts: (i) strict JSON, (ii) Python-literal style dictionaries or lists (e.g., single quotes), and (iii) code-fenced JSON. To avoid silent parsing artifacts, we treat ellipsis-like placeholders (e.g., `"..."` or `[...]` patterns) as invalid and assign zero reward. Parsed tool calls are normalized into a canonical schema $\{\texttt{name}, \texttt{arguments}\}$. Normalization supports common wrappers (e.g., OpenAI-style `"function":{...}`) and converts singleton dict outputs into a length-one list. When arguments appear outside an explicit `"arguments"` field, we fall back to a flat argument map for robustness.

Format reward (r_{fmt}). We use the graded parseability reward defined in Sec. 3.4:

$$r_{\text{fmt}}(\hat{y}) = 0.3 \cdot \mathbb{I}_{\text{tag}} + 0.3 \cdot \mathbb{I}_{\text{parse}} + 0.4 \cdot \mathbb{I}_{\text{norm}}, \quad (7)$$

where $\mathbb{I}_{\text{tag}}$ checks presence of `<tool_call_answer>`, $\mathbb{I}_{\text{parse}}$ checks that the enclosed content parses under the relaxed loader,

and $\mathbb{I}_{\text{norm}}$ checks that normalization yields at least one canonical tool call. This reward stabilizes early training by providing non-zero signal before full functional correctness becomes common.

Accuracy reward (r_{acc}). For function-call correctness, we compute a dense soft-matching reward between predicted calls $\hat{C}$ and gold calls $C^\star$. As described in Sec. 3.4, each gold call is greedily matched to the highest-scoring unused prediction. For a matched pair $(\hat{c}, c^\star)$, we compute: (i) exact tool-name match $s_{\text{name}} \in \{0, 1\}$, (ii) argument-key overlap $s_{\text{key}} \in [0, 1]$ as F1 over key sets, and (iii) value match $s_{\text{val}} \in [0, 1]$ as the fraction of matching values over intersecting keys. The per-pair score is a convex combination:

$$s(\hat{c}, c^\star) = \lambda_{\text{name}} s_{\text{name}} + \lambda_{\text{key}} s_{\text{key}} + \lambda_{\text{val}} s_{\text{val}}, \quad (8)$$

with $(\lambda_{\text{name}}, \lambda_{\text{key}}, \lambda_{\text{val}}) = (0.2,\ 0.3,\ 0.5)$. We average the matched scores over gold calls to obtain a base accuracy score $\bar{s}$.

Robust value comparison. Value matching uses a conservative comparator to reduce sensitivity to superficial formatting. We treat two values as equal if they match exactly, or if they match after numeric coercion and whitespace normalization. To avoid float precision artifacts, long numeric strings are treated as identifiers and compared as normalized strings rather than coerced floats. If neither numeric nor string normalization applies, we fall back to canonical JSON comparison.

Penalty for extra tool calls. To discourage over-prediction and tool-call spamming, we apply a multiplicative penalty for extra predicted calls:

$$r_{\text{acc}} = \bar{s} \cdot \frac{1}{1 + \alpha \cdot \max(0, |\hat{C}| - |C^\star|)}, \quad (9)$$

with $\alpha = 0.25$ (`EXTRA_CALL_PENALTY_ALPHA`). This penalty leaves correct-length predictions unchanged while downweighting completions that append spurious calls.

Table 6 summarizes the fixed reward-related hyperparameters used for **`Solver`** training.

Hyperparameter	Value
Accuracy weights $(\lambda_{\text{name}}, \lambda_{\text{key}}, \lambda_{\text{val}})$	(0.2, 0.3, 0.5)
Extra-call penalty coefficient α	0.25
Format reward weights $(\mathbb{I}_{\text{tag}}, \mathbb{I}_{\text{parse}}, \mathbb{I}_{\text{norm}})$	(0.3, 0.3, 0.4)

Table 6: **Reward hyperparameters for `Solver` training.**

F Further Details on Evaluation

We conduct a comprehensive evaluation across different agentic tool-calling tasks to collectively assess diverse aspects of function invocation. Tool-Alpaca (Tang et al., 2023) examines generalization across heterogeneous tool categories, emphasizing robustness to synthetic distribution shifts. Seal-Tools (Wu et al., 2024) extends this setting to large-scale APIs spanning diverse domains, reducing potential data contamination while stressing scalability. NexusRaven (Srinivasan et al., 2023) focuses on high-fidelity function execution over realistic APIs drawn from enterprise and cybersecurity domains, where precise adherence to function signatures is essential. Differently, API-Bank (Li et al., 2023) evaluates multi-turn scenarios that require models to select appropriate APIs within conversational context. We additionally include SNIPS (Coucke et al., 2018), a spoken language understanding dataset that we adapt to function-calling format, introducing natural language variation absent from synthetic benchmarks. All benchmarks are evaluated using Abstract Syntax Tree matching metric, which verifies structural correctness of function names, parameter presence, and type adherence.

G Training Details of Supervised Baseline Tool-Calling Agents

To contextualize the zero-data effectiveness of *Tool-R0*, we compare against several prominent tool-calling agents that were originally trained on comprehensive, curated supervised datasets. Below, we briefly describe each baseline:

- **xLAM (Zhang et al., 2025c):** xLAM is a family of Large Action Models designed to empower AI agent systems through enhanced function-calling capabilities. It aggregates diverse agent trajectories from various environments and is originally fine-tuned on a unified dataset of approximately 60,000 samples from sources like ToolBench, Webshop, ToolAlpaca, HotpotQA, AlfWorld, APIBank, Mind2Web, AgentBoard, AgentBench, and synthetic datasets such as API-GEN and SpecTools, using supervised fine-tuning (SFT) on models ranging from 1B to 8x22B parameters. For trainings, we used official data provided by authors at Huggingface: `https://huggingface.co/datasets/Salesforce/xlam-function-calling-60k`.

- **Hammer (Lin et al., 2025):** It focuses on robust function-calling for on-device language models via function masking techniques to improve generalization and reduce overfitting. It is originally trained on the augmented xLAM-function-calling-60k dataset that includes around 210,000 samples and 7,500 irrelevance detection samples, using SFT on Qwen 2.0 series models (0.5B to 7B parameters), emphasizing advanced training methods over data refinement. For generating the main training dataset, we used the official codebase provided by the authors and run it for the corresponding data generation resulting in 210,000 samples: https://github.com/MadeAgents/Hammer.

- **ToolACE (Liu et al., 2025c):** An automatic agentic pipeline that generates accurate, multi-turn, and diverse tool-learning data through self-evolution synthesis and multi-agent interactions, creating over 500,000 dialogs based on 26,507 diverse APIs. It employs SFT on Llama-3.1-8B-Instruct, with a dual-layer verification system (rule-based and model-based) to ensure data quality, supporting single, parallel, nested calls, and multi-turn dialogues. We used the publicly available dataset from Huggingface: https://huggingface.co/datasets/Team-ACE/ToolACE.

- **ToolRL (Qian et al., 2025):** A RL framework for tool learning, arguing that reward signals are sufficient without heavy reliance on SFT data curation. It is originally trained on a mixed dataset of 4,000 samples (2,000 from ToolACE, 1,000 from masked Hammer, and 1,000 from xLAM) using GRPO, focusing on reward design for multi-step interactions, irrelevant tool detection, and generalization across unseen scenarios. For trainings we use the official code, main scripts and datasets provided by authors: https://github.com/qiancheng0/ToolRL.

Notably, three of the four baselines we compare against (xLAM, ToolACE, and ToolRL, the last of which is built in part from ToolACE samples) rely on synthesis from strong external LLMs as a core part of their data pipelines: xLAM uses the APIGen pipeline with OpenAI GPT models for both generation and quality filtering, ToolACE uses multi-agent dialogs driven by a frontier LLM together with a model-based verifier, and ToolRL trains on a mixture that inherits these provenances. Hammer is built on top of the xLAM-60k corpus and thus inherits the same dependency. In this light, the comparison in Table 2 is not merely between curated and uncurated training data, but between methods that depend on a strong external LLM as their effective supervisor and *Tool-R0*, which uses only its own base model throughout. The "zero data" claim is therefore stronger than the absence of human annotation alone: *Tool-R0* also operates without any external-LLM-generated supervision, which is the actual cost driver in the baselines.

Since none of these methods release model checkpoints trained on identical base models, a direct comparison of published results would conflate differences in base model capabilities with differences in training data and methodology. To ensure a controlled and fair evaluation, we re-train all baselines on same base model Qwen-2.5-1.5B-Instruct, using the officially released datasets provided by each method's respective authors.

For the three SFT-based baselines (xLAM, Hammer, and ToolACE), we conduct supervised fine-tuning using LLaMA-Factory (Zheng et al., 2024), following the original training configurations as closely as possible (e.g., learning rate schedules, number of epochs, and context length). For ToolRL, we use the official training repository and codebase released by the authors. We train using GRPO with the same Qwen-2.5-1.5B-Instruct, adopting the default hyperparameters specified by the authors main scirpts.

In total, we try to re-implement each baseline faithfully (SFT methods with SFT, and RL methods with RL) on Qwen-2.5-1.5B-Instruct model, using author-provided datasets and closely matched training procedures. This unified setup isolates the effect of training data and algorithmic design from confounding factors such as base model choice, enabling a rigorous assessment of *Tool-R0*'s zero-data effectiveness relative to these data-intensive approaches.

H Additional Experiments and Analysis of *Tool-R0*

This appendix provides extended experimental results and analysis that complement the main paper. While Sec. 4 presents the core findings, here we examine the internal learning dynamics of *Tool-R0* in greater depth; tracing how the `Generator` evolves across self-play iterations, characterizing the co-evolutionary stability between `Generator` and `Solver`, and identifying the practical limits of

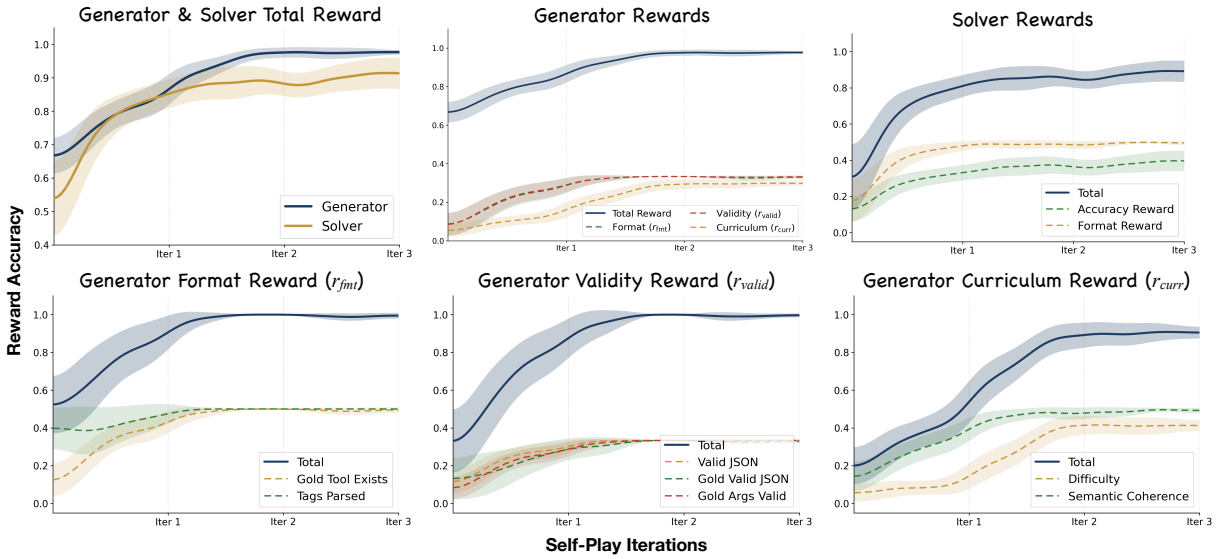

Figure 6: **RLVR training dynamics of `Generator`–`Solver`. Top Row:** total reward trajectories for `Generator` and `Solver` (left), `Generator` total reward with format (r_{fmt}), validity (r_{valid}), and curriculum (r_{curr}) components (middle), and `Solver` total reward with format and accuracy components (right). **Bottom Row:** detailed `Generator` reward components, showing format reward (left; r_{fmt}), validity reward (middle; r_{valid}), and curriculum reward decomposed into difficulty and semantic coherence (right; r_{curr}), over self-play iterations.

self-play scaling. Together, these analyses aim to provide a more complete mechanistic understanding of why *Tool-R0* works, not merely that it does.

The section is organized around five questions. **RQ8** tests whether *Tool-R0* can serve as a mid-training stage that amplifies subsequent supervised post-training. **RQ9** revisits and expands the `Generator` ablation from Sec. 4.2, triangulating its contribution through performance, training signals, and qualitative task evolution. **RQ10** examines the reward dynamics that govern `Generator`-`Solver` co-evolution, drawing connections to adversarial training stability. **RQ11** studies self-play convergence at scale, characterizing when and why improvement saturates across model capacities. Finally, **RQ12** breaks down failure modes of the base model and quantifies the degree to which self-play resolves each category.

Research Question 8: Can *Tool-R0* serve as an effective mid-training strategy to amplify post-training? Recent work shows that mid-training can unlock RL scaling gains invisible from base model evaluations (Wang et al., 2025b). We test whether this extends to tool learning. Each *Tool-R0* iteration serves as a mid-training checkpoint, followed by supervised fine-tuning with ToolACE (Liu et al., 2025c). As shown in Fig. 7, *Tool-R0* + SFT surpasses SFT-only from the first iteration and continues to scale with each subsequent round, ultimately outperforming both SFT-only and

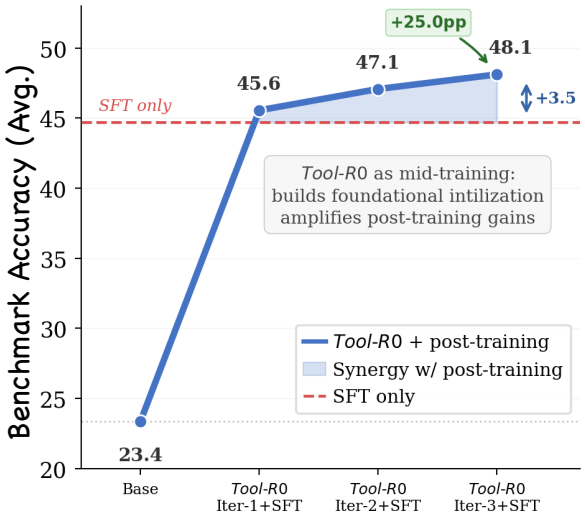

Figure 7: *Tool-R0* as Mid-training.

standalone *Tool-R0*. The implication is practical: self-play builds a stronger foundation from which supervised post-training extracts more from the same human data; the two stages are complementary, not redundant.

Takeaway 5

Tool-R0 serves as an effective mid-training stage that strengthens the foundation for supervised post-training, enabling the model to better explore and extract from the same human-curated data, outperforming human supervision alone.

Research Question 9: Does the Generator meaningfully improve through self-play, and is its learning essential for effective tool-use performance? A core claim of our approach is that the Generator does not merely produce static training data but actively learns to synthesize progressively challenging curricula. We try to validate this claim from three complementary angles: ablation performance, training dynamics, and qualitative illustration.

9.1 Performance Accuracy. To isolate the contribution of Generator learning, we conduct an ablation in which the Generator is frozen after initialization and used only to produce tasks via prompting, while the Solver continues to train. In this setting, the Generator no longer receives optimization signals from the rewards described in Sec. 3.2. As shown in Table 3, freezing the Generator leads to a consistent drop of -6.19 points in average accuracy of Solver.This degradation indicates that performance gains in *Tool-R0* are not solely driven by additional training iterations or static data generation, but critically depend on the Generator's ability to generate increasingly *targeted and informative challenges that align with the Solver's actual learning needs during self-play*.

9.2 Training Signals. Fig. 6 (Bottom Right) tracks the Generator's curriculum reward decomposition across training. The difficulty component rises steeply from 0.1 to 0.4 between Iterations 1–2 before plateauing, indicating the Generator learns to produce harder tasks until reaching the Solver's capacity ceiling. Crucially, semantic coherence remains stable at 0.5 throughout, confirming that increased difficulty does not sacrifice task validity. The total curriculum reward converges near 0.9, reflecting successful joint optimization of both objectives.

9.3 Qualitative Evolution. Fig. 13 and Fig. 14 contrast Generator outputs from early versus late trainings stages. At first iteration of self-play, the Generator produces minimal tasks: single-sentence requests, one available tool with two parameters, and a single-call solution. By Iteration 3, complexity increases across all dimensions—user requests contain five explicit constraints (dates, passenger count, cabin class, hotel location), the tool menu expands to two functions with eleven total parameters, and gold solutions require two coordinated tool calls with cross-task dependencies (flight arrival date must precede hotel check-in). This progression from surface-level to compositional multi-step planning demonstrates learned curriculum generation rather than random variation.

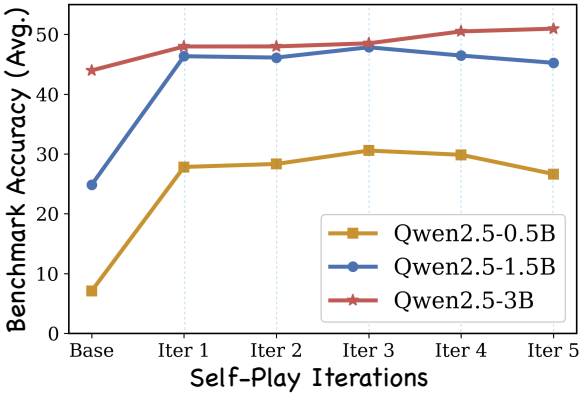

Figure 8: Self-play convergence for extended iterations across model scales.

Research Question 10: What do the reward dynamics reveal about the learning behavior and co-evolutionary stability of Generator–Solver self-play? We examine reward trajectories across self-play iterations to understand the learning dynamics of Generator–Solver co-evolution (Fig. 6). Several patterns emerge: first, the Generator converges faster than the Solver, and Generator total reward reaches $\sim$0.98 by iteration two, while Solver reward stabilizes around 0.90. This asymmetry reflects the inherent difficulty gap-synthesizing valid tasks is easier than solving them. Second, reward components exhibit a clear learning hierarchy. For the Generator, format compliance saturates within the first iteration, validity rewards improve steadily through iteration two, and curriculum rewards show the steepest growth trajectory. This ordering suggests that Generator first learns structural constraints, then internal consistency, and finally calibrating task difficulty. For the Solver, format rewards rise faster than accuracy rewards, with accuracy remaining the performance bottleneck even at convergence. Third, the curriculum reward decomposition reveals stable co-evolution: difficulty increases sharply from 0.1 to 0.5 between iterations one and iteration two as the Generator learns to challenge the improving Solver, yet semantic coherence rises gradually rather than collapsing. This confirms that the Generator produces progressively harder tasks without sacrificing validity. The convergence of both agents toward high total reward with narrowing gap suggests that they both approach a stable equilibrium where Generator output matches Solver capacity—consistent with the saturation behavior observed in downstream evaluation.

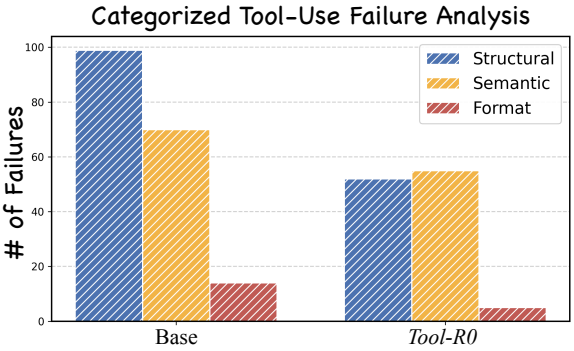

Figure 9: Categorical analysis of tool failures for base model and *Tool-R0*.

Takeaway 6

Reward dynamics reveal a stable and structured co-evolution, reminiscent of adversarial dynamics in GANs (Goodfellow et al., 2014), where the `Generator` learns curriculum shaping faster than the `Solver` learns execution and converges to an equilibrium aligned with solver capacity.

Research Question 11: When does self-play saturate, and what factors limit continued improvement? To understand self-play convergence, we extend training to five iterations and analyze performance trends across model scales. We observe rapid gains after the first iteration, with accuracy typically peaking around the third for smaller models, after which improvements saturate or even slightly degrade. The rapid stabilization in low-capacity models is consistent with early convergence to a Nash-like equilibrium and a potential knowledge boundary, where `Generator` and `Solver` become mutually aligned. In contrast, the 3B model exhibits a more steady and continuous improvement with no signs of saturation, suggesting that higher model capacity delays convergence and that the model has potential to accumulate further gains from additional self-play iterations. This explains the pattern observed in **Research Question 2**: smaller models converge toward their upper bound more quickly, yielding large initial gains but limited headroom for further improvement, while higher-capacity models evolve more gradually but consistently under self-play.

Research Question 12: What are the common tool-use failures of the base model and how does *Tool-R0* address them? To understand where self-play helps at a fine-grained level, we group failures into three categories: *structural* (wrong tool name, incorrect number of calls, extra or missing arguments), *semantic* (wrong argument values, missing gold keys), and *format* (malformed or unparseable JSON). As shown in Fig. 9, the base model fails predominantly on structural errors, the most critical category: wrong tool selection reflects a misunderstanding of tool capabilities, incorrect call counts show an inability to decompose requests into multi-tool plans, and hallucinated arguments indicate poor schema adherence. *Tool-R0* reduces these by nearly half, confirming that iterative self-play strengthens TIR across tool selection, multi-step planning, and schema compliance. Semantic errors also drop but remain the dominant bottleneck, while format errors, already a relative strength of the base model, are reduced to near elimination.

I Full Algorithm of *Tool-R0*

Algorithm 1 presents the complete pseudocode of the *Tool-R0* self-evolution. The algorithm takes a base LLM and a task-specification distribution as input and returns a trained `Solver` policy after K co-evolutionary iterations. Each iteration proceeds through three color-coded stages: **Generator Training** (yellow), where the `Generator` learns to synthesize challenging tasks guided by the frozen `Solver`'s competence frontier; **Dataset Construction** (green), where generated tasks are deduplicated, cross-verified, and organized into a difficulty-based curriculum; and **Solver Training** (blue), where the `Solver` is trained on this curated curriculum with dense accuracy rewards.

J Use of LLMs

In this work, LLMs were used for three purposes: (i) as base models under study for self-play training, (ii) as baselines for empirical comparison, and (iii) for minor assistance in refining the readability of this manuscript. Open-source models (e.g., Qwen (Yang et al., 2025) and Llama (Grattafiori et al., 2024)) were employed as both training subjects and supervised baselines. For writing refinement, the prompt used was similar to *"Please make more clear sentence, making sure to remove any grammatical mistakes."* Importantly, all scientific ideas, methods, experimental design, and conclusions originate entirely from the authors. When LLMs were used for language refinement, all outputs were carefully reviewed to prevent the introduction of hallucinated or incorrect content, ensuring that all arguments, findings, and perspectives are solely those of the authors.

Algorithm 1 *Tool-R0*: Zero-Data Self-Play for Tool-Calling Agents

Require: Base LLM π; iterations K; task-spec distribution $p(s)$ over $s{=}(d, c, m, n)$; probe rollouts M; band $[P_{\text{low}}, P_{\text{high}}]$; width σ

Initialize `Generator` $\pi_\theta^{(0)} \leftarrow \pi$, `Solver` $\pi_\phi^{(0)} \leftarrow \pi$
for $t = 1, \ldots, K$ **do**

Generator Training (Sec. 3.2): train `Generator` with frozen `Solver`

  Freeze `Solver` $\pi_\phi^{(t-1)}$
  for $u = 1, \ldots, U_G$ **do** ▷ GRPO steps
    Sample task specification $s \sim p(s)$
    Sample $x \sim \pi_\theta^{(t-1)}(\cdot \mid s)$ yielding $(q, \mathcal{T}, c^\star)$
    Compute $r_{\text{fmt}}(x)$ ▷ Eq. (1)
    Compute $r_{\text{valid}}(x)$ ▷ Eq. (2)
    Probe frozen `Solver` M times; estimate $\hat{p}_{\text{succ}}$
    Compute $r_{\text{diff}}(x)$ (band-pass on $\hat{p}_{\text{succ}}$) ▷ Eq. (3)
    Compute $r_{\text{sem}}(x)$ (semantic alignment scoring)
    $r_{\text{curr}}(x) \leftarrow r_{\text{diff}}(x) + r_{\text{sem}}(x)$
    $R_G(x) \leftarrow r_{\text{fmt}}(x) + r_{\text{valid}}(x) + r_{\text{curr}}(x)$
  end for
  $\pi_\theta^{(t)} \leftarrow \text{GRPO}\left(\pi_\theta^{(t-1)}, R_G\right)$

Dataset Construction (Sec. 3.3): curate curriculum from frozen `Generator`

  Freeze `Generator` $\pi_\theta^{(t)}$
  Sample candidate pool $\mathcal{P}$ from $\pi_\theta^{(t)}(\cdot \mid s)$
  Deduplicate via canonicalized question–tool–call signatures
  Cross-verify with frozen `Solver` $\pi_\phi^{(t-1)}$; retain consistent tasks
  Estimate difficulty (pass@M) and bucket into {easy, medium, hard}
  Construct curriculum $\mathcal{D}_t$: domain-balanced mix, ordered easy $\rightarrow$ hard

Solver Training (Sec. 3.4): train `Solver` on curated curriculum

  for $u = 1, \ldots, U_S$ **do** ▷ GRPO steps
    Sample minibatch $B \sim \mathcal{D}_t$ (easy $\rightarrow$ hard)
    Sample $\hat{y} \sim \pi_\phi^{(t-1)}(\cdot \mid q, \mathcal{T})$ yielding predicted calls $\hat{C}$
    Compute $r_{\text{fmt}}(\hat{y})$ (tag/parse/normalization)
    Compute $r_{\text{acc}}(\hat{C}, C^\star)$ (name/key/value + extra-call penalty)
    $R_S(\hat{y}) \leftarrow r_{\text{fmt}}(\hat{y}) + r_{\text{acc}}(\hat{C}, C^\star)$
  end for
  $\pi_\phi^{(t)} \leftarrow \text{GRPO}\left(\pi_\phi^{(t-1)}, R_S\right)$
end for
return $\pi_\phi^{(K)}$

Generator Prompt

```
You are an expert task generator for tool-calling agents.

FIRST, in your private scratch-pad, reason step-by-step to design a realistic, non-trivial task that cannot be solved
without correctly calling one or sometimes multiple tools.

CONTROL SPEC (MUST FOLLOW EXACTLY):
- Domain: {domain}
- Context type: {context_type}  (single_turn or multi_turn)
- Number of available tools: {tool_menu_size} (<available_tools>)
- Number of gold tool calls: {num_calls} (<tool_call_answer>)

RULES TO SATISFY THE SPEC:
1) You MUST output exactly {tool_menu_size} tools in <available_tools>.
2) You MUST output exactly {num_calls} tool calls (JSON list length) in <tool_call_answer>.
3) Domain must be {domain}. Do not drift into other domains.
4) If context_type=multi_turn, embed a short conversation in <question> like: "# Conversation\nUser: ...\nAgent: ...\nUser
: ...\nAgent: ..."
5) Tool arguments must be flat primitives only (no lists, no nested objects).
6) The function values (<value1>, <value2>, ...) MUST be present inside user question (<question>...</question>),
otherwise agent cannot solve the task.

THEN, without revealing your reasoning, output the following four blocks in the exact format, NOTHING ELSE:

<think>
Your private reasoning here.
</think>

<question>
Write a natural user question (no bullet points, no meta-instructions, no placeholders).
It must be a natural question, be in domain "{domain}", and mention the exact argument values that appear in <
tool_call_answer>.
</question>

<available_tools>
A JSON list of tools. Each tool MUST include: "name", "description", and "parameters".
[
    {
        "name": "<tool_name>",
        "description": "<short description>",
        "parameters": {
        "<param1>": {"type": "<param1_type>", "description": "<param1_description>"},
        "<param2>": {"type": "<param2_type>", "description": "<param2_description>"},
        ...
        },
        "required": [<param1>, ...],
    },
    ...
]
</available_tools>

<tool_call_answer>
[
{\"name\": \"<tool_name>\", \"arguments\": {\"<param>\": <value>, ...}}
]
</tool_call_answer>

Generate a new tool-calling task now. Follow the CONTROL SPEC exactly and remember to format the output exactly as
instructed.
```

Figure 10: **Generator prompt used for task synthesis in *Tool-R0*.** The system template enforces domain/context constraints, tool-menu cardinality, and primitive-only arguments, while requiring the final output to follow a strict four-block schema (`<think>`, `<question>`, `<available_tools>`, `<tool_call_answer>`).

Solver Prompt

```
You are a strict quality control judge for a synthetic data generation pipeline.
You will be given a User Question, Available Tools, and a Tool Call Answer.

Your job is to score the example on a scale of 1 to 5 based on TWO criteria:
1. **Question Quality**: Is the user question realistic, specific, and clear? (CRITICAL)
2. **Semantic Coherence**: Does the tool call actually solve the user's request?

Scoring Rubric:
- 5 (Perfect): The question is specific and realistic (e.g., "Book a flight to Paris on Dec 5th"). The tool call perfectly
 addresses it.
- 4 (Good): The question is good, but the tool call has minor issues (e.g., slightly different parameter values that still
 work).
- 3 (Passable): The question is vague or simple. The tool call matches it.
- 2 (Bad Question): The question is generic, placeholder text (e.g., "User request here", "Make a tool call"), or nonsense
. **Score 2 or 1 immediately if the question is bad.**
- 1 (Failure): The tool call is completely unrelated, OR the question is clearly a template error (e.g., "A single
concrete user request").

**IMPORTANT:** If the User Question looks like an instruction (e.g., "Generate a query...") rather than a natural user
request, you MUST give a score of 1 or 2.

Reply with ONLY a single integer from 1 to 5.
```

Figure 11: **Semantic alignment reward prompt template.** Prompt used to compute the semantic alignment reward r_{sem}, evaluating whether the synthesized user question is realistic and well-formed, and whether the corresponding tool call semantically and functionally satisfies the user's request.

Solver Prompt

```
A conversation between user and tool-calling assistant. The user asks a question, and the assistant uses tools to solve it
. The assistant first thinks about the reasoning process in the mind and then provides the user with the answer. The
reasoning process and answer are enclosed within <think>...</think> and
<tool_call_answer>...</tool_call_answer> tags, i.e.:
<think>
This is my reasoning.
</think>
<tool_call_answer>
[
  {
    "name": "tool_name",
    "arguments": {"arg1": "value", "arg2": "value2", ...}
  }
]
</tool_call_answer>

User Query:
<question>
USER_QUERY
</question>

Available Tools (JSON):
<available_tools>
TOOL_MENU
</available_tools>
```

Figure 12: **Solver prompt template.** USER_QUERY indicates the user's natural-language input, while TOOL_MENU denotes the set of available tools; both placeholders are instantiated dynamically at each training step.

Early Training (Iter 1): Surface-level User Request

User Request: Book a flight from London to Paris.

Available tools (JSON):

```
[
  {
    "name": "FlightBookingService",
    "description": "A tool to book flights using a travel service.",
    "parameters": {
      "type": "object",
      "properties": {
        "origin": {"type": "string", "description": "Origin airport name"},
        "destination": {"type": "string", "description": "Destination airport name"}
      },
      "required": [origin, destination]
    }
  }
]
```

Output:

```
[
  {
    "name": "FlightBookingService",
    "arguments": {"origin": "LHR", "destination": "CDG"}
  }
]
```

Figure 13: **Early-stage generator behavior in *Tool-R0*.** In the first training iteration, the generator produces surface-level, minimally structured tool-use tasks. The user request is simple and underspecified, the tool menu contains a single available function, and the solution consists of a single, straightforward tool call with canonical arguments.

Late Training (Iter 3): Complex User Request with Multi-step Tool-Calling

User request: What is the best way to book a round-trip from **New York** to **Paris** in **first week of July 2023** for **two passengers** at **business class**, and also **book a hotel in central Paris** for **one night**?

Available tools (JSON):

```
[
  {
    "name": "FlightBooking",
    "description": "A service to book flights.",
    "parameters": {
      "type": "object",
      "properties": {
        "departureCity": {"type": "string", "description": "The city where the flight is departing."},
        "arrivalCity": {"type": "string", "description": "The city where the flight is arriving."},
        "departureDate": {"type": "string", "description": "The date of departure for the round-trip flight."},
        "arrivalDate": {"type": "string", "description": "The date of arrival for the round-trip flight."},
        "passengerCount": {"type": "integer", "description": "The number of passengers for the round-trip flight."},
        "cabinClass": {"type": "string","description": "Cabin class (e.g., economy, premium_economy, business, first)."}
      },
      "required": ["departureCity", "arrivalCity", "departureDate", "arrivalDate", "passengerCount", "cabinClass"]
    }
  },
  {
    "name": "HotelBooking",
    "description": "A service to book hotels.",
    "parameters": {
      "type": "object",
      "properties": {
        "city": {"type": "string","description": "City where the hotel is located."},
        "checkInDate": {"type": "string", "description": "Hotel check-in date."},
        "checkOutDate": {"type": "string", "description": "Hotel check-out date."},
        "guestCount": {"type": "integer", "description": "Number of guests."},
        "locationPreference": {"type": "string", "description": "Preferred area/neighborhood (e.g., 'central Paris', 'city
 center')."
        }
      },
      "required": ["city", "checkInDate", "checkOutDate", "guestCount", "locationPreference"]
    }
  }
]
```

Answer:

```
[
  {
    "name": "FlightBooking",
    "arguments": {"departureCity": "New York", "arrivalCity": "Paris", "departureDate": "2023-06-01", "arrivalDate":
"2023-06-02", "passengerCount": 2, "cabinClass": "business"}
  },
  {
    "name": "HotelBooking",
    "arguments": {"city": "Paris", "checkInDate": "2023-06-02", "checkOutDate": "2023-06-03", "guestCount": 2, "
locationPreference": "central"}
  }
]
```

Figure 14: **Late-stage generator behavior in *Tool-R0*.** By Iteration 3, the generator synthesizes complex, multi-constraint user requests that require multi-step tool execution. The task jointly involves booking a round-trip flight and reserving a centrally located hotel, incorporating temporal constraints, passenger count, and cabin class preferences. The generated solution correctly decomposes the request into multiple ordered tool calls, demonstrating improved task abstraction, constraint integration, and compositional planning.