TIER: Trajectory-Invariant Execution Rewards for Multi-Step Tool Composition

Anay Kulkarni[*]
UC San Diego
ankulkarni@ucsd.edu

ChiaEn Lu[*]
UC San Diego
allenlu@ucsd.edu

Dheeraj Mekala
UC San Diego
dmekala@ucsd.edu

Jayanth Srinivasa
Cisco Research
jasriniv@cisco.com

Gaowen Liu
Cisco Research
gaoliu@cisco.com

Jingbo Shang[†]
UC San Diego
jshang@ucsd.edu

Abstract

Tool use enables large language models to solve complex tasks through sequences of API calls, yet existing reinforcement learning approaches fail to scale to multi-step composition settings. Outcome-based rewards provide only sparse feedback, while trajectory-supervised rewards depend on annotated reference solutions, penalizing valid alternatives and limiting scalability. We propose **TIER**: Trajectory-Invariant Execution Rewards, a reward framework that derives supervision directly from function schemas and runtime execution, rather than from reference trajectories. The reward decomposes into format validity, schema adherence, execution success, and answer correctness, providing graded, interpretable sequence-level feedback derived from verification of each tool call. This design allows any valid execution path to receive credit, naturally supporting multiple solution strategies and requiring no trajectory re-annotation when tool interfaces change. On Depth-Bench, a compositional benchmark stratified by depth (0- to 6-step tasks), TIER achieves $>$90% accuracy across all strata, while trajectory-supervised rewards fall below 25% from 2 steps onward and reach 0% at 5–6 steps. We further demonstrate consistent gains on benchmarks such as BFCL v3 and NestFUL. Ablations show that schema and execution rewards are each necessary, highlighting the importance of multi-level supervision for compositional reasoning. Code and datasets are available at https://github.com/anaykulkarni/TIER.

1 Introduction

Large language models (LLMs) have shown strong capabilities in tool use, enabling interaction with external environments through structured API calls [Schick et al., 2023, Qin et al., 2024]. Building on the success of RL with verifiable rewards for reasoning [Guo et al., 2025], recent work applies RL to tool use [Qian et al., 2025, Zhang et al., 2026], but these methods primarily target single-step tool invocation or aggregate success metrics. Tasks that require composing multiple tool calls, however, remain challenging: performance degrades rapidly with depth, which limits the deployment of LLM-based agents in real-world settings.

Tool composition refers to tasks where solutions require multiple interdependent tool calls. For example, answering *"What is the weather at the destination of my next flight?"* requires first retrieving the flight information, extracting its destination, and only then querying the weather. We refer to the minimum number of tool calls required for a task as its *depth*, and call a depth-d task a d*-step* task.

[*]Equal contribution.
[†]Corresponding author.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

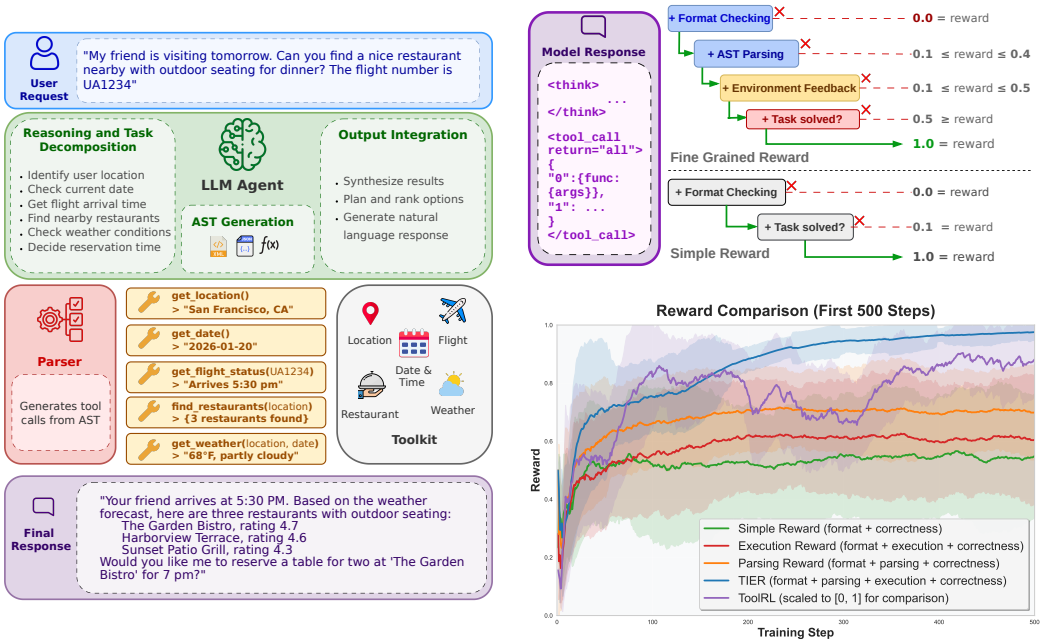

Figure 1: **Left:** Overview of TIER for tool-augmented LLM agents. The model structures tool calls as abstract syntax trees (ASTs), which are executed against external tools. Unlike trajectory-supervised approaches, rewards are computed directly from function schemas and execution outcomes. **Right (Top):** TIER provides fine-grained, multi-level supervision decomposed into format, schema, execution, and answer correctness. In contrast, simple outcome rewards offer only sparse feedback. **Right (Bottom):** Training curves under different reward designs. Only the full fine-grained TIER reward achieves stable convergence to high performance. Partial rewards have wide variance bands, and their mean reward plateaus early at a lower value. Trajectory-supervised rewards (ToolRL) are unstable: the mean reward oscillates without settling, due to inconsistent supervision across alternative valid compositions. Lines are exponentially weighted moving averages of the per-step batch-mean reward (smoothing constant $\alpha = 0.01$); shaded regions are ± 1 standard deviation of the unsmoothed reward over a 50-step rolling window. For visualization, we linearly rescale ToolRL's reward from $[-3, 4]$ to $[0, 1]$.

We demonstrate that under simple outcome rewards, accuracy can drop from over 93% at depth 1 to 7% at depth 2 and collapse to 0% from depth 4 onward (Table 1). A key challenge lies in reward design. Outcome-based reinforcement learning with verifiable rewards (RLVR) provides only a sparse signal upon task completion, which makes it difficult to assign credit to intermediate decisions such as tool selection, argument specification, and execution order [Shao et al., 2024, Guo et al., 2025]. Recent works introduce more fine-grained supervision by comparing model outputs to human-curated reference trajectories [Qian et al., 2025, Zeng et al., 2025]. However, trajectory-based rewards penalize valid alternative solutions, require costly annotation, and are vulnerable to changes in tool interfaces. These limitations compound in compositional settings, where multiple valid execution paths typically exist.

We propose TIER (Figure 1), a reward framework for tool learning that derives supervision directly from function schemas and runtime execution rather than reference trajectories. Each tool call is evaluated through a structured process that verifies format validity, schema adherence, execution success, and answer correctness, yielding graded, interpretable feedback at the sequence level. Because the verifier reads schemas directly at training time, supervision tracks schema changes automatically; unlike trajectory-supervised approaches [Qian et al., 2025], no reference trajectories need to be regenerated when tools are added, modified, or deprecated. Any valid solution path receives appropriate credit, admitting multiple correct compositions that are typical of multi-step tasks. Empirically, TIER achieves >90% accuracy across all composition depths on DepthBench and improves on external benchmarks including BFCL v3 and NestFUL.

Our contributions are as follows:

- We introduce TIER, a trajectory-invariant reward framework that leverages function schemas and execution feedback, eliminating the need for annotated reference trajectories.
- We introduce DepthBench, a depth-stratified benchmark spanning depths 0 (abstention) to 6, enabling systematic evaluation of compositional generalization.
- We show that reward components are complementary and not interchangeable, emphasizing that graded, multi-level supervision is essential for higher composition depths. We support our claims with gains on DepthBench and external benchmarks (BFCL v3, NestFUL).

We publish DepthBench, the training data, and code, to support reproducibility and facilitate future research on compositional tool use: `https://github.com/anaykulkarni/TIER`

2 TIER: Trajectory-Invariant Execution Rewards

TIER is built on a simple principle: rewards for tool-use trajectories should depend on *what the sequence accomplishes*, not *which path it follows*. This section formalizes this notion of trajectory invariance and then introduces the four reward components that enable it.

2.1 Design principle: trajectory invariance

A reward function R is *trajectory-invariant* if its value on a candidate sequence y depends only on properties verifiable from y and the tool environment—format validity, schema adherence, execution success, and answer correctness—and not on y's similarity to any reference trajectory y^*.

Compositional tool use requires two further properties of a trajectory-invariant reward. (i) *Path invariance*: every sequence that passes all verifiable checks receives the maximum reward, regardless of which path was taken: alternative orderings of independent sub-goals, commutative operations, and functionally equivalent APIs all receive equal credit. (ii) *Graded feedback*: sequences that fail are differentiated by which verifiable properties they violate, providing a learning signal that does not depend on a reference trajectory. Trajectory-supervised rewards lack path invariance—they arbitrarily privilege one valid path—while outcome-based rewards are path-invariant but lack graded feedback, collapsing all error states into one. Section 4.2 shows that both properties are necessary.

TIER also has an operational benefit: because rewards are derived from schemas and runtime execution, supervision tracks tool changes automatically. When tools are added, modified, or deprecated, no reference trajectories need to be re-annotated as long as final answers remain valid.

2.2 Reward definition

The TIER reward is computed once per generated tool-call sequence as

$$R_{\text{total}} = R_{\text{format}} + R_{\text{parse}} + R_{\text{exec}} + R_{\text{answer}}, \quad \text{with } R_{\text{total}} \text{ scaled to } [0, 1]$$

Each component verifies a distinct property of the full sequence—syntactic, structural, operational, and semantic—and is computed without reference to ground-truth trajectories. We represent tool sequences as structured ASTs that expose call ordering, argument bindings, and any nesting structure.

Format Validity. $R_{\text{format}} \in \{0, 1\}$ checks whether the full AST is well-formed and parsable. If this fails, all downstream rewards are zero.

Schema Adherence (Parsing). Decomposes as

$$R_{\text{parse}} = R_{name} + R_{param} + R_{dtype}, \quad R_{\text{parse}} \in [0, 3]$$

reflecting three granularities of schema correctness. Tool names are checked categorically

$$R_{name} = \begin{cases} 1 & \text{if all tool names are valid} \\ 0 & \text{otherwise} \end{cases}$$

Because name validity determines which schema each call is checked against, any invalid API makes the sequence unverifiable and by extension invalid ($R_{\text{parse}} = 0$). Parameter and type correctness are graded by mismatch counts,

$$R_{param/dtype} = \text{clip}(1 - \lambda_p \cdot p, 0, 1),$$

where p is the total number of mismatches across all calls in the sequence, each call's parameters checked against its own schema. We set $\lambda_p = 0.25$, balancing two failure modes: too large a

coefficient eliminates partial credit for nearly correct schemas, while too small a coefficient makes individual mismatches negligible.

Execution Success. $R_{\text{exec}} \in \{0, 1\}$. Equals 1 iff every tool call executes successfully; otherwise 0. The all-or-nothing rule reflects the structure of compositional dependence: a failure at step k invalidates downstream calls that depend on its output, so a graded reward (e.g., fraction of successful calls) would credit work that produced no usable result.

Answer Correctness. $R_{\text{answer}} = 5 \cdot 1[\text{answer correct}] \in \{0, 5\}$. We weight answer correctness substantially higher than the other components because lower weightings induce reward hacking—the model satisfies format, schema, and execution validity without producing correct answers, and training stalls before the answer correctness signal can take effect. The weighting prioritizes end-task success while preserving graded shaping signal from the lower-level components.

How the reward satisfies both properties. Each component is computed once over the full sequence rather than accumulated per step. Per-step accumulation would let longer sequences collect more reward and would require a per-step target, which in practice comes from a reference trajectory; scoring the sequence as a whole keeps the maximum reward independent of length and of the order of independent sub-goals, which gives path invariance. Graded feedback comes from the decomposition itself: a sequence that fails is scored by which components it still satisfies (Section 2.1).

2.3 Reinforcement learning

We train models using a GRPO-style policy gradient objective with group-normalized advantages and token-level loss aggregation. Given a prompt x, we sample a set of trajectories $\mathcal{G}(x) = \{y^{(1)}, \ldots, y^{(K)}\}$ from the current policy $y \sim \pi_\theta(\cdot \mid x)$ and execute each to obtain scalar rewards $R(x, y) \in [0, 1]$.

Group-normalized advantages. Advantages are computed at the trajectory level by normalizing rewards within each rollout group:

$$A(x, y) = \frac{R(x, y) - \mu_{\mathcal{G}(x)}}{\sigma_{\mathcal{G}(x)} + \varepsilon},$$

where $\mu_{\mathcal{G}(x)}$ and $\sigma_{\mathcal{G}(x)}$ are the mean and standard deviation of rewards over $\mathcal{G}(x)$, and $\varepsilon = 10^{-4}$.

Policy gradient objective. We optimize a PPO-style clipped surrogate objective at the token level. For a trajectory $y = (y_1, \ldots, y_{T_y})$:

$$\rho_t(\theta) = \frac{\pi_\theta(y_t \mid x, y_{<t})}{\pi_{\theta_{\text{old}}}(y_t \mid x, y_{<t})},$$

$$\mathcal{L}_{\text{PG}}(\theta) = -\frac{1}{\sum_{(x,y)\in\mathcal{B}} T_y} \sum_{(x,y)\in\mathcal{B}} \sum_{t=1}^{T_y} \min\Big(\rho_t(\theta)A(x, y),\ \text{clip}(\rho_t(\theta), 1 - \epsilon_{\text{clip}}, 1 + \epsilon_{\text{clip}})\, A(x, y)\Big),$$

where $\mathcal{B}$ denotes the local batch, T_y is the number of active (non-padding) tokens in trajectory y, and $\epsilon_{\text{clip}} = 0.2$. The loss is normalized by the total number of active tokens in the batch.

In our setup, we take a single policy update per batch of rollouts, so $\pi_{\theta_{\text{old}}} = \pi_\theta$ at the update step and $\rho_t(\theta) = 1$. In this case, the clipped surrogate reduces in gradient to the standard REINFORCE estimator [Williams, 1992] $-A(x, y)\nabla_\theta \log \pi_\theta(y_t \mid x, y_{<t})$.

KL regularization. We apply per-token KL regularization to a reference policy π_{ref} using the k3 estimator [Schulman, 2017]:

$$\hat{\text{KL}}_t = \exp\big(\log \pi_{\text{ref}}(y_t \mid \cdot) - \log \pi_\theta(y_t \mid \cdot)\big) - \big(\log \pi_{\text{ref}}(y_t \mid \cdot) - \log \pi_\theta(y_t \mid \cdot)\big) - 1,$$

$$\mathcal{L}(\theta) = \mathcal{L}_{\text{PG}}(\theta) + \frac{\lambda_{\text{KL}}}{\sum_{(x,y)\in\mathcal{B}} T_y} \sum_{(x,y)\in\mathcal{B}} \sum_{t=1}^{T_y} \hat{\text{KL}}_t, \quad \text{with } \lambda_{\text{KL}} = 0.04$$

The reward formulation in Section 2.2 is agnostic to the optimization algorithm. We verify this by evaluating three loss-aggregation variants of GRPO under TIER: standard GRPO, batch-normalized GRPO, and DAPO-style token-level aggregation [Yu et al., 2025], observing comparable performance (Appendix H).

3 DepthBench: A depth-stratified benchmark for multi-step tool composition

3.1 Environment

DepthBench evaluates multi-step tool composition over a fully executable backend of 163 hand-crafted tools spanning three categories: information and utility functions, structured retrieval over synthetic datasets, and multi-step workflows. All tools execute deterministically, supporting both reproducible evaluation and the runtime grading needed to compute TIER's execution and answer rewards. A rule-based parser validates schema adherence on each call, serving as the evaluation harness at test time and the R_{parse} verifier during training. To evaluate transfer beyond this controlled setting, we additionally benchmark on BFCL v3 [Patil et al., 2025] and NestFUL [Basu et al., 2025].

3.2 DepthBench design and evaluation protocol

DepthBench contains 1,710 queries stratified by composition depth from 0 to 6, where depth 0 denotes irrelevance cases in which no available tool can answer the query, and depths 1–6 denote single-call through six-call compositions. Multi-call compositions (depths 2–6) account for 42% of the benchmark; the remainder consists of single-call queries and irrelevance cases. Composition structures include linear chains ($A \to B \to C$), funnels ($(A, B) \to C$), trees ($A \to (B, C)$), and order-agnostic conjunctive structures ($A \to (B \& C \& D) \to E$). See Appendix C.

Although DepthBench is smaller than prior tool-use benchmarks, its construction is deliberately controlled: tool schemas are modeled after commonly used APIs with executable backends implemented to simulate their behavior; queries are hand-crafted and verified against their intended composition structure. Comparable benchmarks make different trade-offs: neither BFCL v3, which relies largely on AST matching, nor NestFUL (1,861 examples) is stratified by depth, leaving DepthBench as the only setting we are aware of that combines realistic, executable tool semantics with depth-stratified evaluation. This stratification, absent from prior work [Patil et al., 2025, Basu et al., 2025, Zhong et al., 2025], enables systematic analysis of compositional generalization rather than aggregate success.

Train–validation splits use disjoint sets of target tools, with distractors also disjoint between splits, ensuring that no tool seen at training time appears at evaluation time in any role. Distractors are deliberately chosen to share schemas and output types with target tools, requiring models to distinguish functionally similar APIs rather than relying on superficial cues such as names or signature patterns. For multi-step evaluation, the dataset is split into 944 training and 766 validation examples, with tool schemas and sampled distractors provided at both training and evaluation time. Each query is annotated with its expected final answer and a single gold tool-call trajectory. TIER uses only the expected answer; the gold trajectory serves the trajectory-supervised baseline (ToolRL). See Appendix A for the dataset construction, per-depth counts (Table 5), and depth stratification rationale.

4 Experiment Results

4.1 Experimental setup

Model and Training. All experiments start from Qwen3-8B. We train on a context length of 12,288 tokens with the batch-normalized GRPO objective [Shao et al., 2024], i.e., the loss of Section 2.3, normalized over the active tokens of the local batch, regularized against a fixed reference policy (KL penalty $\lambda_{\text{KL}} = 0.04$). For each prompt, we sample $K = 8$ trajectories and compute group-normalized advantages (Section 2.3). Unless stated otherwise, TIER's reward formulation from Section 2.2 is held fixed; Section 4.3 provides ablations over individual reward components. To isolate the effect of reward design from optimization, we additionally evaluate other loss aggregations (standard GRPO, DAPO-style) under the same reward in Appendix H. Training rollouts sample with temperature 1.0 and top-p 1.0; evaluation samples with temperature 0.6 and top-p 0.95, and the intervals in Tables 1 and 7 reflect this evaluation sampling variance.

Baselines. We compare TIER against representative alternatives: (1) *Simple (outcome-based)*, a sparse outcome reward based only on format validity and final answer correctness, with no intermediate supervision; (2) *ToolRL (trajectory-supervised)* [Qian et al., 2025], which uses fine-grained rewards derived from matching reference trajectories; (3) *Qwen3-8B (zero-shot)*, Qwen3-8B without further training; (4) *Qwen3-8B (3-shot ICL)*, the same model given three chaining demonstrations, each

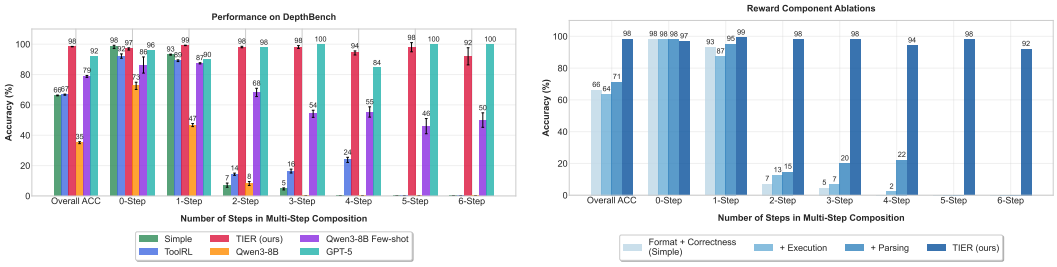

Figure 2: **(a)** Performance on DepthBench; "Qwen3-8B" and "Qwen3-8B Few-shot" in the legend are the zero-shot and 3-shot ICL rows of Table 1. **(b)** Reward component ablations under TIER.

Table 1: Performance comparison on DepthBench, stratified by composition depth. RL and Qwen3-8B values are mean $\pm$ 95% Student-t CI over $N = 10$ evaluation runs of a single trained model; GPT-5 is a single run. RL arms and the zero-shot row receive no in-context demonstrations; the 3-shot ICL rows receive three chaining demonstrations, each illustrating a different composition structure.

Model / Reward	Training	Overall Acc	0-Step	1-Step	2-Step	3-Step	4-Step	5-Step	6-Step
Simple	RL	66.23 $\pm$0.26	**98.24 $\pm$1.03**	93.12 $\pm$0.32	7.05 $\pm$1.37	4.67 $\pm$0.67	0.00 $\pm$0.00	0.00 $\pm$0.00	0.00 $\pm$0.00
ToolRL	RL	66.75 $\pm$0.36	92.16 $\pm$1.43	89.10 $\pm$0.56	14.32 $\pm$0.73	16.40 $\pm$1.28	23.78 $\pm$1.68	0.00 $\pm$0.00	0.00 $\pm$0.00
TIER (ours)	RL	**98.39 $\pm$0.15**	96.86 $\pm$0.72	**99.19 $\pm$0.16**	**98.11 $\pm$0.48**	**98.13 $\pm$0.92**	**94.22 $\pm$1.34**	**98.00 $\pm$3.02**	**92.00 $\pm$5.64**
Qwen3-8B	Zero-shot	35.10 $\pm$0.65	72.55 $\pm$2.38	46.69 $\pm$0.99	8.21 $\pm$1.32	0.00 $\pm$0.00	0.00 $\pm$0.00	0.00 $\pm$0.00	0.00 $\pm$0.00
Qwen3-8B	3-shot ICL	78.76 $\pm$0.63	86.27 $\pm$5.32	87.38 $\pm$0.39	68.21 $\pm$2.73	54.13 $\pm$2.34	55.33 $\pm$3.39	46.00 $\pm$5.00	50.00 $\pm$4.77
GPT-5*	3-shot ICL	92.17	96.08	89.79	97.89	100.00	84.44	100.00	100.00

illustrating a different composition structure; and (5) *GPT-5 (3-shot ICL)*, a frontier proprietary model under the same 3-shot protocol, included as a reference rather than a head-to-head comparison.

All RL arms, including ToolRL, share the training split, rollout budget, optimizer, hyperparameters, and hardware, and are evaluated with identical prompts and sampling; only the reward differs.

Evaluation Benchmarks. We evaluate on three benchmarks: (1) *DepthBench* (Section 3), evaluated on the 766-example validation set, with results stratified by composition depth (0–6); (2) *BFCL v3* [Patil et al., 2025], a widely used function-calling benchmark with different tools and evaluation protocols that measures out-of-distribution generalization; and (3) *NestFUL* [Basu et al., 2025], which evaluates nested and sequential API composition, where we report a controlled comparison against ToolRL under Qwen3-8B, and a broader comparison with tool-use models trained by other mechanisms. Finally, to evaluate whether DepthBench's compact, composition-focused supervision is competitive with substantially larger tool-use corpora, we additionally train Qwen3-8B under TIER on RL-adapted versions of ToolACE [Liu et al., 2025] and xLAM [Zhang et al., 2024] (Appendix G).

4.2 Multi-step composition requires trajectory invariance and execution awareness

We test the central claim that, among RL reward designs, only TIER sustains multi-step composition. We report DepthBench accuracy stratified by composition depth along with population-weighted average of per-depth accuracies in Table 1, Figure 2a. Three observations follow:

(1) TIER sustains high performance across all depths, achieving >90% accuracy at every composition depth, including depths 5 and 6, where queries combine several operators (e.g., funnels and trees) into longer compositions. Verbatim rollouts are presented in Section E.1 of the Appendix.

(2) Outcome-based rewards fail beyond shallow composition. Simple-RL drops from 93.12% at one step to 7.05% at two steps and to 0% from 4 steps onward. Because the reward assigns the same value to every well-formed incorrect solution, it provides no gradient toward partially correct compositions; the policy instead collapses to single-call outputs (Table 7). Its training reward has wide variance bands, and its mean plateaus early at a low value (Figure 1).

(3) Trajectory supervision collapses for a different reason. Although ToolRL provides intermediate feedback, it penalizes valid alternative compositions that do not match the reference trajectory, and

*GPT-5 results represent frontier capability at the time of writing, and not a head-to-head comparison.

Table 2: Effect of reward granularity on the validation split. The Simple and TIER rows are means over 10 evaluation runs (Table 1); the two partial-reward rows are single runs.

Reward	Overall Acc	0-Step	1-Step	2-Step	3-Step	4-Step	5-Step	6-Step
Format + Correctness (Simple)	66.23	98.24	93.12	7.05	4.67	0.00	0.00	0.00
+ Execution (no Parsing)	63.58	98.04	87.29	12.63	6.67	2.22	0.00	0.00
+ Parsing (no Execution)	71.28	98.04	95.21	14.74	20.00	22.22	0.00	0.00
TIER (full reward)	**98.39**	96.86	99.19	98.11	98.13	94.22	98.00	92.00

it reaches 0% accuracy at depths 5 and 6, where multiple valid paths are common. At these depths, its failed attempts fall short of the required chain length (Table 7). Its training reward oscillates without settling (Figure 1), consistent with valid solutions receiving low reward whenever they deviate from the reference. We attribute the collapse to a misspecified reward rather than to training length; Section E.2 gives examples of ToolRL mis-scoring both valid and broken compositions.

Depth-0 measures abstention. Depth-0 items fail for reasons unrelated to chain construction, most commonly by invoking a plausible but inapplicable tool, so we treat them as a separate axis rather than the shallow end of the depth ordering. All RL methods score 92–98% there and we do not interpret their ordering; both Qwen3-8B prompting baselines score well below (Table 1). Section 4.4 tests abstention out of domain, where the methods separate.

Non-monotonicity. Accuracy over depths 1–6 does not decrease strictly for every method, for two reasons. First, the bins at depths 5–6 are small (Appendix A), so intervals widen; most adjacent-depth inversions (TIER 2→3 and 4→5; 3-shot ICL 3→4 and 5→6) fall within overlapping 95% intervals, and we do not interpret them. The only inversion with non-overlapping intervals is ToolRL's rise from depth 2 to 4 (14.32 → 23.78; Appendix F). Second, depth is not the sole measure of difficulty. Composition structure also contributes to the difficulty of a question. A depth-5 fan-out over independent calls with one aggregation is easier than a depth-4 chain where each call's arguments bind to the previous return value, since only the latter propagates errors. Adjacent bins are therefore not guaranteed to be difficulty-ordered. The claim Table 1 supports is the between-method comparison at a fixed depth, not the within-method ordering across depths.

RL versus prompting. The zero-shot row of Table 1 is the prompt-matched reference for every RL arm: RL models receive no demonstrations, while the 3-shot ICL rows add three demonstrations, each illustrating a different composition structure. Simple-RL (66.23%) and ToolRL (66.75%) nearly double zero-shot accuracy (35.10%), but the gain comes almost entirely from depths 0–1; at depth 2, Simple-RL (7.05%) is no better than the zero-shot model (8.21%). Both fall below 3-shot ICL (78.76%): under these rewards, RL acquires less compositional skill than three demonstrations convey. TIER, trained and evaluated without demonstrations, exceeds the 3-shot Qwen3-8B baseline at every depth. TIER's accuracy does not come from extra tool calls: although the grader permits superfluous calls, every solved depth-d query contains exactly d calls under every method (Appendix F).

4.3 Reward-component ablation

While Table 1 establishes that TIER enables multi-step composition, it does not isolate which components are necessary. We ablate the reward by removing components individually, holding the model, optimizer, rollout budget, and RL algorithm fixed. Format and answer correctness are retained in all settings as the minimal baseline (identical to Simple-RL). When a component is removed, the remaining components keep their original weights and the total is rescaled by its new maximum to $[0, 1]$, so every configuration shares the same reward range. Results are shown in Table 2, Figure 2b.

Components are complementary, not additive. Partial rewards fail abruptly beyond shallow depths. Simple drops to 0% from depth 4 onward; adding either execution or parsing yields modest improvement at intermediate depths but still collapses to 0% at depths 5 and 6. Only the full reward sustains >90% across all depths. The transition from 0% to 90% at six steps under identical training setup represents a qualitative shift in training outcomes rather than gradual improvement: the components are not substitutes whose contributions sum incrementally, but a set whose union unlocks compositional behavior.

A counterintuitive result: adding execution reward without parsing decreases overall accuracy, with 1-step accuracy also dropping. We inspect failures by reviewing rollout logs at 100-step

Table 3: Comparison across different training objectives on BFCL. Results are from a single run. Overall is the official BFCL score.

Reward format	Prompt style	Overall	Non-Live AST	Live	Multiturn	Relevance	Irrelevance
TIER	Ours	**68.92**	85.94	**81.96**	**39.12**	77.78	87.65
Simple	Ours	66.30	85.00	78.72	34.00	77.78	83.56
ToolRL	Ours	37.27	84.10	43.98	0.38	**94.44**	0.81
Supervised Fine Tuning	Ours	61.47	84.12	79.52	19.25	61.11	**90.33**
Qwen3-8B	Ours	63.20	83.58	77.39	28.25	83.33	83.18
Qwen3-8B	Qwen default	64.31	**88.81**	78.54	33.38	77.78	79.08

intervals throughout training. We observe a reward-hacking pattern: in the absence of schema-level discrimination from parsing, the model learns to route most queries to a small set of "safe" tools that reliably execute, accumulating execution reward while answering incorrectly. Up-weighting answer correctness prevents this collapse only when parsing supplies a discriminative signal; on its own, it is insufficient. This supports the rationale of designing parsing and execution as complementary components: parsing enforces structural validity (correct tool signature) and execution enforces operational validity (no runtime error). Neither suffices alone.

Failure is due to signal saturation, not instability. The training curves in Figure 1 (bottom right) show that the mean reward of every partial-reward configuration plateaus early, below the optimum. Their variance bands are wide, but unlike ToolRL's, their mean reward does not oscillate. The model saturates the available signal; the signal itself lacks the structure needed for compositional behavior. Only the full reward reaches near-ceiling performance.

Implications. Multi-step composition benefits from concurrent supervision at four levels: syntactic (format), structural (schema), operational (execution), and semantic (answer). Errors at each level produce different failure modes: format errors block parsing entirely, schema errors propagate to downstream calls, execution errors break the chain at runtime, and semantic errors corrupt outputs without raising any lower-level flag. The fully decomposed reward, however, addresses error states at their origin. Compositional tool learning is therefore not a single objective but a multi-level credit-assignment problem requiring reward signals targeted at distinct failure modes.

4.4 Transfer to external benchmarks

We evaluate out-of-domain transfer on BFCL v3. We create a new 1,240-example training split of DepthBench, distinct from the 944/766 split; since evaluation is on BFCL, it also draws on DepthBench validation queries. We vary only the reward mechanism and also include an SFT baseline (Appendix I) and Qwen3-8B without further training for reference. For BFCL's long-context tasks, we use YaRN [Peng et al., 2024] to extend the context length of all models to 128K. TIER achieves the best overall accuracy (Table 3).

The gain is not the prompt. The trained arms and the Qwen3-8B "Ours" row use our system prompt (Appendix D) without demonstrations, so Qwen3-8B under our prompt (63.20) is the prompt-matched reference for every trained arm. Prompting alone does not close the gap: Qwen3-8B scores 64.31 with Qwen's default BFCL prompt and 63.20 with our prompt, whose format is out of distribution for the untrained model. Holding our prompt fixed, TIER training raises accuracy from 63.20 to 68.92, and the four training arms span 31.65 points (TIER 68.92, Simple 66.30, SFT 61.47, ToolRL 37.27), a spread the prompt cannot explain.

Where the gains concentrate. TIER's reward verifies schema adherence and argument binding, properties of the call rather than of the corpus, so gains should appear where these are load-bearing. On Irrelevance, TIER reaches 87.65 against 83.18 for the matched reference, while ToolRL collapses to 0.81; with its high Relevance (94.44), this indicates a model that calls a tool unconditionally, the expected outcome of trajectory supervision on references that nearly always contain a call. TIER improves Irrelevance while keeping Relevance close to the matched reference, which we attribute to the depth-0 stratum and a reward that can verify a non-call as correct. On Multiturn, TIER (39.12) improves most over the matched reference (28.25; Simple 34.00, SFT 19.25, ToolRL 0.38) despite single-turn training: stale or hallucinated bindings break multi-turn tasks, and execution grounding penalizes exactly these. On Live, TIER reaches 81.96 against 77.39 for the matched reference. SFT,

Table 4: **NestFUL results. (a)** Without demonstrations, training Qwen3-8B with TIER outperforms ToolRL. **(b)** Under the 3-shot prompting setup of Basu et al. [2025], TIER with Qwen3-8B compared against other tool-learning methods. Exact Match Accuracy (EM Acc) is the ratio of test cases where all predicted APIs execute correctly and match the ground truth exactly. Our results are from a single run; baseline results are from Basu et al. [2025].

(a) TIER vs. ToolRL

Method	EM Acc
ToolRL	0.476
TIER	**0.684**

(b) Comparison with tool-use models

	Hammer2.1-7B	TIER	xLAM-8x7b-fc-r	DeepSeek-V3	GPT-4o
# Param	7B	8B	46.7B	685B	UNK
EM Acc	0.08	**0.75**	0.03	0.60	0.60

trained on verified rollouts (one per query; Appendix I), falls below the matched reference overall (61.47 vs. 63.20), consistent with overfitting to a single target trajectory.

NestFUL. We evaluate on NestFUL under two protocols. In a controlled comparison without demonstrations (Table 4(a)), TIER achieves a 0.684 exact-match accuracy versus ToolRL's 0.476. To compare against published results (Table 4(b)), we follow the 3-shot prompting setup of the original paper, under which the reported baselines were evaluated; TIER's accuracy rises to 0.75, outperforming DeepSeek-V3 (685B, 0.60) and GPT-4o (0.60). The difference between the two TIER numbers reflects the prompting protocol, not a different model. SFT-based models such as xLAM and Hammer2.1 achieve near-zero accuracy despite strong single-call accuracy, consistent with SFT alone being insufficient for composition, though base models and data differ.

4.5 Intermediate representation for tool calls

Tool interaction requires a structured intermediate representation (IR) that can be deterministically parsed and executed. We evaluate JSON-based, and XML-based ASTs along with direct generation without an IR under TIER (Table 6, Figure 3, Appendix B). JSON achieves the best overall performance and is therefore used as the default IR in all experiments. Since structured IRs for tool use are already well established in prior work, we defer implementation details to the appendix.

5 Related work

Tool Use via Prompting and Supervised Learning. Early systems rely on prompting-based methods such as Chain-of-Thought and ReAct [Wei et al., 2022, Yao et al., 2023], as well as engineered pipelines that interleave reasoning and tool calls [Paranjape et al., 2023, Zhuang et al., 2024, Song et al., 2023]. More recent works fine-tune models on curated tool-use trajectories or schemas [Schick et al., 2023, Liu et al., 2024, Liang et al., 2024], improving robustness but requiring annotated intermediate actions and fixed tool interfaces. Other work trains models to follow abstract function-calling schemas that generalize across tools via schema injection or retrieval [Qin et al., 2024], enabling plug-and-play extensibility. However, these approaches rely on supervised objectives or exact-match evaluation and do not address learning from execution feedback.

RL for Tool Use. Recent work applies RL as a post-training paradigm for tool learning [Qian et al., 2025, Zhang et al., 2026, Zeng et al., 2025, Du et al., 2025]. Outcome-based rewards rely solely on final task success, providing sparse feedback that does not assign credit across intermediate decisions. Concurrent work such as PORTool [Wu et al., 2025] addresses this by redistributing trajectory-level rewards using rollout-tree comparisons, but the resulting signal remains path-dependent and tied to specific exploration trajectories. ToolRL [Qian et al., 2025] introduces fine-grained supervision, but derives it by matching model outputs to annotated reference trajectories, requiring curated data and penalizing valid alternative solutions. Nemotron-Research-Tool-N1 [Zhang et al., 2026] uses a binary rule-based reward that combines format validity with a match of each call against the ground-truth call; like ToolRL, it scores against a reference rather than execution, and it does not evaluate composition by depth. In contrast, TIER provides fine-grained trajectory-invariant supervision from function schemas and runtime execution, without relying on reference trajectories.

Multi-Step Tool Composition. Despite extensive work on tool learning, systematic evaluation and learning of multi-step composition remains limited. Existing benchmarks such as NestFUL [Basu et al., 2025] and ComplexFuncBench [Zhong et al., 2025] evaluate compositional tasks but do not stratify by composition depth. Prior RL approaches do not demonstrate reliable generalization across varying composition lengths. DepthBench addresses this gap directly by enabling per-depth accuracy

measurement, which we show is essential for diagnosing reward failure modes. We demonstrate that TIER enables reliable composition where both outcome-based and trajectory-supervised rewards fail.

6 Conclusion and future work

We show that reward design is the primary bottleneck for multi-step tool composition in RL-trained language models. Outcome-based rewards fail at deeper compositions due to sparse credit assignment, while trajectory-supervised rewards penalize valid alternative solutions, a limitation that worsens as the number of feasible execution paths grows with composition depth. TIER addresses both issues by deriving rewards deterministically from function schemas and runtime execution, enabling fine-grained supervision that is invariant to the chosen solution path. Reward-component ablations further show that compositional tool use benefits from supervision at the syntactic, structural, operational, and semantic levels simultaneously; removing schema or execution supervision substantially degrades performance. TIER achieves $>90\%$ accuracy across all composition depths on DepthBench and generalizes consistently to BFCL v3 and NestFUL.

An important future direction is extending TIER to multi-turn RL. By feeding model outputs from previous turns back into subsequent turns, the model can iteratively diagnose and correct failures. Since rewards are execution-grounded, both success and failure states can be incorporated automatically into the training signal, making it well-suited for scalable multi-turn training, where constructing such datasets is otherwise prohibitive. Appendix K describes this further. Another promising direction is applying TIER's underlying RLVR principle to domains such as code generation and NL-to-SQL, where correctness can also be verified deterministically, enabling RL training without trajectory supervision. Finally, the depth stratification in DepthBench suggests a natural curriculum-learning strategy: training can begin with shallow compositions and progressively introduce deeper ones as the model's capabilities improve. Such a curriculum may make deeper compositions easier to learn.

7 Limitations

Our primary benchmark uses a synthetic, deterministic backend to enable controlled analysis of reward design, but real-world tool environments are stochastic and constrained, with latency, noisy outputs, partial failures, rate limits, monetary cost, and irreversible side effects. TIER assumes exploratory execution is cheap, repeatable, and safe, which may not hold in practice. Although BFCL v3 and NestFUL provide initial evidence of generalization, evaluating TIER in stochastic, cost-sensitive, and safety-critical settings, and training it with multi-turn RL rather than the single-turn training used here, remains future work.

Furthermore, all experiments use a single model—Qwen3-8B—so we do not study generalization across model families or scaling across sizes. Extending to other model families was limited by compute: because our call format is out of distribution for pretrained models, the Llama and Mistral models we tested required substantial SFT warm-starting before RL became viable, whereas Qwen3-8B could be trained with RL from a cold start. All training arms share this backbone, so relative comparisons are internally valid, and the loss-aggregation ablation (Appendix H) suggests the gains arise primarily from reward design.

Reward weights are fixed based on preliminary tuning, and sensitivity to different weighting schemes across domains and composition depths is not explored. Finally, because DepthBench does not contain long-context function calling scenarios, training is limited to a 12,288-token context. Consequently, we evaluate BFCL's long-context tasks with YaRN context extension (Section 4.4) but exclude ComplexFuncBench [Zhong et al., 2025], which targets long-context scenarios throughout.

8 Impact Statement

This work advances post-training methods for large language models with an emphasis on reward design. The proposed approach reduces dependence on costly human annotation for tool-use training, which may broaden access to capable tool-using agents. As with any advance in autonomous agent capabilities, deployment in safety-critical or high-stakes environments requires appropriate guardrails, monitoring, and human oversight. We do not identify ethical concerns specific to this work beyond those generally associated with the development and deployment of language model agents.

References

Kinjal Basu, Ibrahim Abdelaziz, Kiran Kate, Mayank Agarwal, Maxwell Crouse, Yara Rizk, Kelsey Bradford, Asim Munawar, Sadhana Kumaravel, Saurabh Goyal, et al. NESTFUL: A benchmark for evaluating LLMs on nested sequences of API calls. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 33526–33535, 2025.

Weihua Du, Hailei Gong, Zhan Ling, Kang Liu, Lingfeng Shen, Xuesong Yao, Yufei Xu, Dingyuan Shi, Yiming Yang, and Jiecao Chen. Generalizable end-to-end tool-use RL with synthetic CodeGym. *arXiv preprint arXiv:2509.17325*, 2025.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Honghui Ding, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jingchang Chen, Jingyang Yuan, Jinhao Tu, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaichao You, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingxu Zhou, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature*, 645(8081):633–638, September 2025. ISSN 1476-4687. doi: 10.1038/s41586-025-09422-z.

Yaobo Liang, Chenfei Wu, Ting Song, Wenshan Wu, Yan Xia, Yu Liu, Yang Ou, Shuai Lu, Lei Ji, Shaoguang Mao, et al. TaskMatrix.AI: Completing tasks by connecting foundation models with millions of APIs. *Intelligent Computing*, 3:0063, 2024.

Weiwen Liu, Xu Huang, Xingshan Zeng, Xinlong Hao, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Zhengying Liu, Yuanqing Yu, Zezhong Wang, Yuxian Wang, Wu Ning, Yutai Hou, Bin Wang, Chuhan Wu, Xinzhi Wang, Yong Liu, Yasheng Wang, Duyu Tang, Dandan Tu, Lifeng Shang, Xin Jiang, Ruiming Tang, Defu Lian, Qun Liu, and Enhong Chen. ToolACE: Winning the points of LLM function calling. In *The Thirteenth International Conference on Learning Representations*, 2025.

Zuxin Liu, Thai Hoang, Jianguo Zhang, Ming Zhu, Tian Lan, Juntao Tan, Weiran Yao, Zhiwei Liu, Yihao Feng, Rithesh RN, et al. APIGen: Automated pipeline for generating verifiable and diverse function-calling datasets. *Advances in Neural Information Processing Systems*, 37:54463–54482, 2024.

Zixian Ma, Weikai Huang, Jieyu Zhang, Tanmay Gupta, and Ranjay Krishna. m&m's: A benchmark to evaluate tool-use for multi-step multi-modal tasks. In *European Conference on Computer Vision*, pages 18–34. Springer, 2024.

Bhargavi Paranjape, Scott Lundberg, Sameer Singh, Hannaneh Hajishirzi, Luke Zettlemoyer, and Marco Tulio Ribeiro. ART: Automatic multi-step reasoning and tool-use for large language models. *arXiv preprint arXiv:2303.09014*, 2023.

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

Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. YaRN: Efficient context window extension of large language models. In *The Twelfth International Conference on Learning Representations*, 2024.

Cheng Qian, Emre Can Acikgoz, Qi He, Hongru Wang, Xiusi Chen, Dilek Hakkani-Tür, Gokhan Tur, and Heng Ji. ToolRL: Reward is all tool learning needs. In *Advances in Neural Information Processing Systems*, 2025.

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

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

John Schulman. Approximating KL divergence. `http://joschu.net/blog/kl-approx.html`, 2017.

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

Yifan Song, Weimin Xiong, Dawei Zhu, Wenhao Wu, Han Qian, Mingbo Song, Hailiang Huang, Cheng Li, Ke Wang, Rong Yao, et al. RestGPT: Connecting large language models with real-world RESTful APIs. *arXiv preprint arXiv:2306.06624*, 2023.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3-4):229–256, 1992.

Feijie Wu, Weiwu Zhu, Yuxiang Zhang, Soumya Chatterjee, Jiarong Zhu, Fan Mo, Rodin Luo, and Jing Gao. PORTool: Tool-use LLM training with rewarded tree. *arXiv preprint arXiv:2510.26020*, 2025.

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

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Weinan Dai, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Mu Qiao, Yonghui Wu, and Mingxuan Wang. DAPO: An open-source LLM reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025.

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

Jianguo Zhang, Tian Lan, Ming Zhu, Zuxin Liu, Thai Hoang, Shirley Kokane, Weiran Yao, Juntao Tan, Akshara Prabhakar, Haolin Chen, Zhiwei Liu, Yihao Feng, Tulika Awalgaonkar, Rithesh Murthy, Eric Hu, Zeyuan Chen, Ran Xu, Juan Carlos Niebles, Shelby Heinecke, Huan Wang, Silvio Savarese, and Caiming Xiong. xLAM: A family of large action models to empower AI agent systems, 2024.

Shaokun Zhang, Yi Dong, Jieyu Zhang, Jan Kautz, Bryan Catanzaro, Andrew Tao, Qingyun Wu, Zhiding Yu, and Guilin Liu. Nemotron-Research-Tool-N1: Exploring tool-using language models with reinforced reasoning. In *The Fourteenth International Conference on Learning Representations*, 2026.

Lucen Zhong, Zhengxiao Du, Xiaohan Zhang, Haiyi Hu, and Jie Tang. ComplexFuncBench: Exploring multi-step and constrained function calling under long-context scenario, 2025.

Yuchen Zhuang, Xiang Chen, Tong Yu, Saayan Mitra, Victor Bursztyn, Ryan A Rossi, Somdeb Sarkhel, and Chao Zhang. ToolChain*: Efficient action space navigation in large language models with A* search. In *The Twelfth International Conference on Learning Representations*, 2024.

A DepthBench Construction

Tools. DepthBench comprises 163 hand-authored tools modeled on commonly used public APIs, each backed by a deterministic executable implementation. Executability is required both for reproducible evaluation and for the runtime grading on which TIER's execution and answer rewards depend. The tools span ten everyday domains (car buying, restaurant booking, food ordering, travel booking, and others), each exposing several verb-level operations so that queries exercise realistic within-domain composition (search $\rightarrow$ filter $\rightarrow$ act) rather than isolated calls.

Why hand-authored. LLM-generated corpora (ToolACE, xLAM) and BFCL contain recurring defects: mismatched schemas, arguments, and tool documentation, and golden answers whose correctness is unverifiable. These are tolerable for trajectory-matching objectives but disqualifying for an execution-grounded reward, since a golden call that violates its own schema causes the schema and answer checks to fire against the reference itself.

Query construction. Each query is derived from a single-call task by progressively removing information, forcing intermediate tool calls to recover the missing inputs. A one-call query supplies everything the call needs ("Find restaurants in New York": `get_restaurants_by_location("New York")`). Removing part of that information yields a two-call query, which can only be expressed as a simple chain ("Find restaurants near me": `get_user_location()` $\rightarrow$ `get_restaurants_by_location()`). Three-call queries introduce one of three operators: *chain* (an additional call on the two-step result), *funnel* (independent calls feeding a final one), or *tree* (branching off intermediate results). Deeper queries recursively combine these operators, often yielding mixed structures rather than a single composition type. We explain these structures in further detail in Appendix C. Depth is labeled as the minimum number of calls required to solve the query.

Validation and contamination. Every example was reviewed independently by both first authors plus a third external reviewer. No LLM generation or LLM-based filtering was used at any stage. Contamination does not apply in the usual sense, as all queries and tools are freshly hand-authored; functionality overlaps conceptually with public benchmarks, but we deliberately altered signatures and schemas so that no tool shares a schema with BFCL or NestFUL.

Table 5: DepthBench composition depth distribution.

Depth	0	1	2	3	4	5	6	Total
Queries	101	890	150	259	190	70	50	1,710
Share	5.9%	52.0%	8.8%	15.1%	11.1%	4.1%	2.9%	100%

Counts and splits. The benchmark is split 944 train / 766 validation; all numbers in Table 1 are reported on the 766-example validation set, stratified by depth.

The stratification is intentional. The $\sim 58\%$ of queries at depths 0–1 establishes the prerequisites for composition: recognizing when a tool is required, when none is, and how to invoke one correctly. A benchmark weighted only toward deep chains cannot separate a model that composes correctly from one that composes indiscriminately, and models trained without that mass over-call. Depths 3–4 carry the largest share among multi-step depths (26.3%) because chaining behavior is acquired there; depths 5–6 (7.0%) require less volume because composition is recursive — a six-step query is the same operator applied more times, not a distinct skill.

B AST-Based Intermediate Representations

We evaluate three strategies for function-call generation: JSON-based abstract syntax trees (ASTs), XML-based ASTs, and direct generation of tool calls without an intermediate representation.

These strategies impose different structural constraints and levels of syntactic flexibility, which affect training stability and execution correctness.

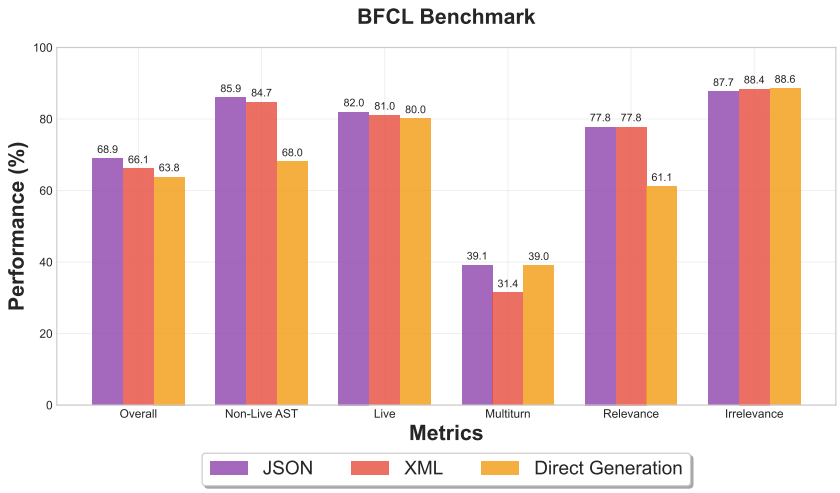

Figure 3: Performance of different intermediate representations under TIER on BFCL v3.

Table 6: Comparison between generation strategies

IR format	Overall	Non-Live AST	Live	Multiturn	Relevance	Irrelevance
JSON	**68.92**	**85.94**	**81.96**	**39.12**	**77.78**	87.65
XML	66.06	84.73	81.03	31.37	**77.78**	88.37
Direct Generation	63.80	68.02	80.05	39.00	61.11	**88.56**

As shown in Table 6, JSON-based IRs achieve the strongest overall performance, outperforming both XML and direct generation across most metrics, including Non-Live AST, Live, Multiturn, and overall accuracy. These results are consistent with observations by [Ma et al., 2024, Patil et al., 2025].

Direct generation performs competitively on certain subsets—matching JSON on Multiturn (39.00 vs 39.12) and slightly outperforming both structured representations on Irrelevance (88.56). However, it degrades substantially in syntactic correctness (Non-Live AST: 68.02 vs 85.94 for JSON), leading to lower overall performance.

XML improves over direct generation in syntactic correctness and overall accuracy, but this advantage is not uniform across metrics: it underperforms direct generation on Multiturn (31.37 vs 39.00) and achieves comparable but not superior performance on Irrelevance.

Overall, structured IRs provide clear benefits for schema adherence and execution reliability, with JSON offering the best trade-off between structural consistency and downstream performance. Based on these results, we adopt JSON as the default intermediate representation in subsequent experiments.

B.1 XML-Based AST Format.

The XML-based IR represents a structured abstract syntax tree (AST) that explicitly encodes tool invocation order and inter-call dependencies:

AST

```
<api id="0">
    <name>API_NAME_0</name>
    <param name="P_0" type="string">VALUE_0</param>
    <param name="P_1" type="integer">VALUE_1</param>
</api>

<api id="1">
    <name>API_NAME_1</name>
    <param name="P_0">
        <response api_id="0" api_name="API_NAME_0" type="
            placeholder"/>
    </param>
</api>
```

The execution order is encoded through explicit API identifiers, while inter-call dependencies are captured via response references, enabling deterministic execution of composed tool calls.

B.2 JSON-Based AST Format.

We additionally support a JSON-based AST format, where tool calls are represented as an ordered mapping from call identifiers to function invocations:

AST

```
{
  "0": {
    "API_NAME_0": {
      "PARAM_NAME_0": VALUE_0,
      "PARAM_NAME_1": VALUE_1
    }
  },
  "1": {
    "API_NAME_1": {
      "PARAM_NAME_0": "API_RESPONSE_0"
    }
  }
}
```

In this format, dependencies between calls are expressed by referencing prior API responses. The parser validates JSON well-formedness, function names, parameter schemas, and data types before execution.

B.3 Direct Call List Format.

For simpler tool-use scenarios, we also support a direct call format, where tool invocations are expressed as a comma-separated list within brackets:

AST

```
[
  API_NAME_0(PARAM_NAME_0=VALUE_0,
             PARAM_NAME_1=VALUE_1),
  API_NAME_1(PARAM_NAME_0=API_RESPONSE_0)
]
```

This format provides a lightweight representation for sequential tool calls without explicit AST structure. The parser infers execution order from list position and resolves dependencies by matching referenced outputs.

C Function Composition Examples

We illustrate function composition of varying depths using representative examples. Here, the number of steps refers to the minimum number of tool calls needed to accomplish the task.

C.1 Linear chaining

Here, all calls are sequentially dependent, forming a chain $A \rightarrow B \rightarrow C$.

Prompt

Get all car listings from nearby dealerships.

AST

```
{
  "0": {
    "Get_current_location": {}
  },
  "1": {
    "Find_Dealerships_by_Location": {
      "location": "API_RESPONSE_0"
    }
  },
  "2": {
    "Get_Car_Listing_by_Dealerships": {
      "dealerships": "API_RESPONSE_1"
    }
  }
}
```

C.2 Funnel Structure

In this setting, intermediate calls are independent and jointly inform the final call, following the structure $(A, B) \rightarrow C$.

Prompt

What is the current weather?

AST


```
{
  "0": {
    "Get_current_time": {}
  },
  "1": {
    "Get_current_location": {}
  },
  "2": {
    "Get_weather": {
      "time": "API_RESPONSE_0",
      "location": "API_RESPONSE_1"
    }
  }
}
```


C.3 Tree Structure

This structure requires a single call that is further branched out into multiple parallel calls $A \rightarrow (B, C)$.

Prompt

What is the price of bitcoin on Jan, 11, 2026. Give me its value in British pounds, and in Euros?

AST


```
{
  "0": {
    "Get_Crypto_Price": {
        "ticker": "BTC",
        "price_time": "2026-01-11"
    }
  },
  "1": {
    "Currency_conversion_API": {
        "amount": "API_RESPONSE_0",
        "from_currency": "USD",
        "to_currency": "GBP",
        "price_time": "2026-01-11"
    }
  },
  "2": {
    "Currency_conversion_API": {
        "amount": "API_RESPONSE_0",
        "from_currency": "USD",
        "to_currency": "EUR",
        "price_time": "2026-01-11"
    }
  }
}
```


C.4 Conjunctive Constraint Composition

This scenario is a constraint-based composition over a shared entity, where multiple tool calls impose independent predicates on the same object.

Prompt

Show me Japanese restaurants in San Diego that have ratings above 4.2.

Option A:

AST

```
{
  "0": {
    "Find_Restaurants_by_Location": {
        "location": "San Diego"
    }
  },
  "1": {
    "Filter_Restaurants_by_Cuisine": {
        "restaurants": "API_RESPONSE_0",
        "cuisine": "Japanese"
    }
  },
  "2": {
    "Filter_Restaurants_by_ratings": {
        "restaurants": "API_RESPONSE_1",
        "ratings": 4.2
    }
  }
}
```

Option B:

AST

```
{
  "0": {
    "Find_Restaurants_by_Location": {
        "location": "San Diego"
    }
  },
  "1": {
    "Filter_Restaurants_by_ratings": {
        "restaurants": "API_RESPONSE_0",
        "ratings": 4.2
    }
  },
  "2": {
    "Filter_Restaurants_by_Cuisine": {
        "restaurants": "API_RESPONSE_1",
        "cuisine": "Japanese"
    }
  }
}
```

Both orderings are valid because the filters commute.

C.5 Mixed structure

Multiple structures can be combined into a single query that requires more complex composition.

Prompt

Could you find Mexican restaurants near me that are open right now?

AST

```
{
  "0": {
    "Get_current_location": { }
  },
  "1": {
    "Find_Restaurants_by_Location": {
        "location": "API_RESPONSE_0"
    }
  },
  "2": {
    "Filter_Restaurants_by_Cuisine": {
        "restaurants": "API_RESPONSE_1",
        "cuisine": "Mexican"
    }
  },
  "3": {
    "Get_current_time": { }
  },
  "4": {
    "Filter_Restaurants_by_Opening_Hours": {
        "restaurants": "API_RESPONSE_2",
        "time": "API_RESPONSE_3"
    }
  }
}
```

D System Prompt

We show the system prompt used for the JSON AST format.

System Prompt

```
# Instruction

You are an intelligent agent capable of advanced tool use. You observe your
    environment, reason deeply about it, and use tools or APIs at your
    disposal to answer user queries. Queries may require single or multiple
    API calls. Multiple calls may be executed in parallel (returning all
    outputs) or chained sequentially (returning only the final output).

## API Call Scenarios

1. **Single API call**: The query requires execution of a single API whose
    output will be returned to the user. (one output)

2. **Multiple Parallel API calls**: The query requires execution of a set of
    API calls simultaneously, and the outputs of all will be returned to the
    user. (all outputs)
```

```
3. **Multiple Chained API calls**: The query requires APIs to be chained
    together, where the output of one API serves as the input to another, and
    only the output of the final API call in the chain is returned to the
    user. (one output)

4. **Multi-step API calls (Single Query)**: A single query requires multiple
    sequential API calls. You respond with ONE API call at a time and wait
    for the environment to execute it and provide you with the tool response.
    If the tool response indicates that the problem is solved, return a
    NO-CALL response. If the tool call fails, retry it. If the tool call
    succeeds, proceed with the next call in the sequence.

5. **Multi-turn Conversations**: An extended conversation where the user
    provides multiple queries across several turns. Each turn may require
    multi-step API calls (as described in scenario 4). You must retain and
    utilize contextual information from previous exchanges to handle
    follow-up queries effectively. After completing one user's request with a
    NO-CALL response, the user may follow up with related queries or entirely
    new tasks. Use information from previous turns when relevant to the
    current query.

## Your Objectives

1. Determine whether the query can be solved using the APIs available to you.
    If not, inform the user accordingly with a NO-CALL response.
2. If the query can be solved, determine whether it requires a single API
    call, multiple parallel calls, multiple chained calls, or multi-step
    sequential calls.
3. Identify the appropriate API(s) from the available list and prepare a plan
    for solving the problem.
4. Determine the correct order of execution when chaining or sequencing is
    required.
5. Construct the final solution using the exact JSON format provided below.

## Response Format

You must strictly respond in one of the following JSON formats:

### Format 1: Single API Call

<think>
Explain your thought process to solve the question.
</think>
<tool_call return="one">
    {
        "0": {
            "API_NAME": {
                "PARAM_NAME_0": PARAM_VALUE_0,
                "PARAM_NAME_1": PARAM_VALUE_1
            }
        }
    }
</tool_call>


### Format 2: Multiple Parallel API Calls

<think>
Explain your thought process to solve the question.
</think>
<tool_call return="all">
    {
```

```
        "0": {
            "API_NAME_0": {
                "PARAM_NAME_0": PARAM_VALUE_0
            }
        },
        "1": {
            "API_NAME_1": {
                "PARAM_NAME_0": PARAM_VALUE_0
            }
        }
    }
</tool_call>


### Format 3: Multiple Chained API Calls

<think>
Explain your thought process to solve the question.
</think>
<tool_call return="one">
    {
        "0": {
            "API_NAME_0": {
                "PARAM_NAME_0": PARAM_VALUE_0
            }
        },
        "1": {
            "API_NAME_1": {
                "PARAM_NAME_0": PARAM_VALUE_0,
                "PARAM_NAME_1": "API_RESPONSE_0"
            }
        }
    }
</tool_call>


### Format 4: Multi-step API Calls (One at a Time)
For multi-step scenarios, respond with ONLY ONE API call per response:

<think>
Explain the current step in your plan.
</think>
<tool_call return="one">
    {
        "0": {
            "API_NAME": {
                "PARAM_NAME_0": PARAM_VALUE_0
            }
        }
    }
</tool_call>

After receiving the tool response, proceed with the next step or return
    NO-CALL if complete.

### Format 5: NO-CALL Response
Use this format when:
- The query cannot be solved using available APIs
- The problem has already been solved based on previous tool responses
- No further action is required

<think>
```

```
Explain why no API calls are required. Specify either: (1) the query cannot be
    solved using the available APIs, or (2) the problem has been solved based
    on the environment response.
</think>
<tool_call return="one">
    {}
</tool_call>


# Rules

1. **API Usage**: Use only the provided APIs. Do not invent, assume, or
    reference APIs that are not explicitly listed.

2. **Return Attribute**: Indicate whether the output of all APIs needs to be
    returned or only one output needs to be returned using the "return"
    attribute of the `<tool_call>` tag:
   - `<tool_call return="all">`: All outputs will be returned to the user. Use
    this for parallel calls.
   - `<tool_call return="one">`: Only the output of the final API call will be
    returned to the user. Use this for single, chained, or multi-step calls.

3. **Parameters**: Include all required parameters in each API call. Optional
    parameters may be added if they improve the result or are necessary to
    fulfill the query.

4. **API IDs**: Each API call must have a unique numeric string key. The keys
    should be zero-indexed (starting from "0").

5. **ID Sequencing**: The keys used should be a sequence of increasing
    integers like "0", "1", "2", "3", etc. For chained calls, this sequence
    represents the order of execution.

6. **JSON Structure**: The tool call JSON must be a dictionary where each key
    is a numeric string ID and each value is an object containing a single
    API name as key and its parameters object as value.

7. **API Name**: Each API call object (e.g., `{"0": {...}}`) must contain
    exactly one API name as the key, with its parameters as the value.

8. **Parameter Format**: Parameters for each API should be provided as a JSON
    object with parameter names as keys and their values as the corresponding
    values.

9. **Parameter Types**: API parameter values may be primitive types (boolean,
    integer, float, string), complex types (dict, array), or string
    placeholders referencing previous API responses.

10. **Response References**: To reference the output of a previous API call in
    chained scenarios, use the string value `"API_RESPONSE_X"` where X is the
    numeric ID of the API whose output you want to reference.

11. **Chaining APIs**: When chaining API calls together, use
    `"API_RESPONSE_0"`, `"API_RESPONSE_1"`, etc. as parameter values to pass
    the output from one API as input to another.

12. **Multi-step Execution**: For multi-step scenarios (scenarios 4 and 5),
    respond with only ONE API call at a time (a JSON object with a single key
    "0"). Wait for the environment to execute it and provide feedback before
    proceeding with the next step.
```

```
13. **Context Retention**: For multi-turn conversations (scenario 5), maintain
    awareness of previous exchanges and use that context when relevant to the
    current query.

14. **NO-CALL Format**: When no API calls are needed, respond with an empty
    JSON object `{}` inside the `<tool_call return="one">` tag.
```

The available tool definitions are added at the end of this prompt dynamically. In addition to the required tools we also add irrelevant and ambiguous tool definitions to elicit reasoning capabilities of the model.

E Traces from the trained model

E.1 Verbatim rollouts

Prompt

Make reservation at HTL_LONDON_PARK for 7 nights from August 1 to August 8, 2024 for 3 people. Guest Robert Wilson, phone 555-0567, email rob@vacation.com. Charge credit card 3782-822463-10005, billing address 789 Pine St, Los Angeles for Robert Wilson.

Model Response

```
<think>
Okay, let's tackle this user request. The user wants to make a hotel
    reservation at HTL_LONDON_PARK for 7 nights from August 1 to August 8,
    2024, for 3 people. The guest details are Robert Wilson, phone 555-0567,
    email rob@vacation.com. They also want to charge a credit card:
    3782-822463-10005, with billing address 789 Pine St, Los Angeles.
First, I need to check which API to use. The available APIs are GetDate,
    BookHotel, Doordash.place_order, and
    BeanAndBrewCoffeehouse.get_opening_hours. The task is about booking a
    hotel, so the BookHotel API is the right choice.
Looking at the parameters required for BookHotel: hotel_id, number_of_nights,
    check_details (with check_in and check_out dates), number_of_people,
    personal_details (name, phone, email), and billing_details (name,
    payment_method, and credit info if applicable).

The user provided all the necessary details. The check-in date is August 1,
    2024, and check-out is August 8, 2024. The number of nights is 7, which
    matches the date difference. The hotel_id is HTL_LONDON_PARK. Number of
    people is 3. Personal details include Robert Wilson, phone 555-0567, and
    email rob@vacation.com. Billing details require the credit card info and
    address. The payment method is credit, so the credit parameters need to
    be included.

Wait, the credit card number is given as 3782-822463-10005. The API might
    require it without hyphens, but the user provided it with hyphens.
    However, the parameter type is a string, so including the hyphens should
    be okay. The billing address is 789 Pine St, Los Angeles, and the billing
    name is Robert Wilson.

I need to structure the parameters correctly. The check_details should have
    check_in as "2024-08-01" and check_out as "2024-08-08". The
    number_of_nights is 7. The personal_details name is Robert Wilson,
    phone_number is 555-0567, and email is rob@vacation.com. The
    billing_details name is Robert Wilson, payment_method is "credit", and
```

```
    the credit object includes address, card_number, and city. The city is
    Los Angeles.

Wait, the billing address is 789 Pine St, Los Angeles. So the address field in
    the credit object should be "789 Pine St, Los Angeles", and the city is
    "Los Angeles". That makes sense. The card number is 3782-822463-10005,
    which I'll input as-is since it's a string.

Now, verifying all required parameters are present. The hotel_id is provided,
    number_of_nights is 7, check_details with both dates, number_of_people is
    3, personal_details with name and phone, and billing_details with all
    required fields. All required parameters are covered.

So, the BookHotel API call should include all these parameters. There's no
    need for other APIs here. The user isn't asking for dates, ordering food,
    or coffeehouse hours. So this single API call should handle the
    reservation.
</think>

<tool_call return="one">
{
  "0": {
    "BookHotel": {
      "hotel_id": "HTL_LONDON_PARK",
      "number_of_nights": 7,
      "check_details": {
        "check_in": "2024-08-01",
        "check_out": "2024-08-08"
      },
      "number_of_people": 3,
      "personal_details": {
        "name": "Robert Wilson",
        "phone_number": "555-0567",
        "email": "rob@vacation.com"
      },
      "billing_details": {
        "name": "Robert Wilson",
        "payment_method": "credit",
        "credit": {
          "address": "789 Pine St, Los Angeles",
          "card_number": "3782-822463-10005",
          "city": "Los Angeles"
        }
      }
    }
  }
}
</tool_call>
```

Prompt

Can you find open French restaurants in my vicinity?

Model Response

```
<think>
Okay, let's see. The user is asking for open French restaurants in their
    vicinity. First, I need to figure out what APIs to use here.
```

Hmm, the available APIs include GetCurrentLocation, which can get the user's location. Then there's FindRestaurantsByLocation, which uses a location parameter to find restaurants. But the user wants French cuisine, so maybe after getting the restaurants, I need to filter by cuisine using FilterByCuisine. Also, they want open restaurants, so I might need to check the opening hours with FilterByOpeningHours. But wait, the current time is needed for that. So first, get the current time with GetTime. Then, get the location, then find restaurants in that location, filter by French cuisine, then check which are open now. But the user might not have provided a specific time, so maybe the opening hours are checked against the current time. Let me outline the steps:

1. Get current time using GetTime.
2. Get current location using GetCurrentLocation.
3. Use FindRestaurantsByLocation with the location from step 2 to get all restaurants there.
4. Filter those restaurants to French cuisine using FilterByCuisine.
5. Then, filter the French restaurants by opening hours using FilterByOpeningHours, using the current time from step 1.

Wait, but the FilterByOpeningHours requires a specific time. So after getting the current time, I can pass that as the time parameter. So the steps are sequential. Each step depends on the previous one's output. So this would be a chained API call scenario. Let me check the parameters.
For example, FindRestaurantsByLocation needs a location, which comes from GetCurrentLocation. Then FilterByCuisine needs the cuisine (French) and the restaurants DataFrame from FindRestaurantsByLocation. Then FilterByOpeningHours needs the current time and the filtered restaurants from FilterByCuisine. So the order is important. So the first call is GetCurrentLocation, then FindRestaurantsByLocation with that location, then FilterByCuisine with cuisine=French and the restaurants from the previous step, then FilterByOpeningHours with time from GetTime and the restaurants from the previous filter.
Then, the final result would be the open French restaurants. But the user might not have specified a time, so the opening hours check is necessary. However, if the current time is not provided, the FilterByOpeningHours can't be done. Therefore, the steps need to be in that order. So the first API call should be GetCurrentLocation to get the location. Then FindRestaurantsByLocation with that location. Then FilterByCuisine for French. Then FilterByOpeningHours with the current time. But the current time needs to be obtained first. Wait, the GetTime API can be called first to get the current time. Then GetCurrentLocation. Then FindRestaurantsByLocation. Then FilterByCuisine. Then FilterByOpeningHours using the time from GetTime. So the order is: GetTime, GetCurrentLocation, FindRestaurantsByLocation, FilterByCuisine, FilterByOpeningHours. But the user's query is to find open French restaurants, so the final step would be FilterByOpeningHours.
However, the user might not have a specific time, so the current time is used. Therefore, the correct sequence is to first get the current time, then the location, then find restaurants in that location, filter by French cuisine, then filter those by opening hours using the current time. So the API calls would be chained in that order. Let me check the parameters again. For example, the FilterByOpeningHours requires the time and the restaurants DataFrame. So after getting the French restaurants, we pass them to FilterByOpeningHours with the current time. So the steps are sequential. Therefore, the first API call should be GetTime to get the current time. Then GetCurrentLocation. Then FindRestaurantsByLocation with the location. Then FilterByCuisine with cuisine=French. Then FilterByOpeningHours with the time from GetTime and the restaurants from the previous step. So the final answer would be the filtered restaurants that are open now. But the user's query is to find open French

```
    restaurants, so the last step is necessary. Therefore, the correct
    approach is to chain these APIs in the order mentioned. So the first call
    is GetTime, then GetCurrentLocation, then FindRestaurantsByLocation, then
    FilterByCuisine, then FilterByOpeningHours. Each step's output is used as
    input for the next. Therefore, the tool calls would be in sequence. Let
    me structure this as a multi-step API call scenario. The first step is to
    get the current time. Then the location. Then find restaurants by
    location. Then filter by cuisine. Then filter by opening hours. So the
    initial call would be GetTime, then GetCurrentLocation, and so on. But
    the user might not have specified a time, so the current time is
    necessary.
Therefore, the first API call is GetTime. Then, the next is
    GetCurrentLocation. Then, using the location from GetCurrentLocation,
    call FindRestaurantsByLocation. Then, filter those with FilterByCuisine
    for French. Then, use the current time from GetTime to filter by opening
    hours. So the final answer would be the list of open French restaurants.
    Therefore, the correct sequence of API calls is GetTime,
    GetCurrentLocation, FindRestaurantsByLocation, FilterByCuisine,
    FilterByOpeningHours. Each step depends on the previous one's output. So
    the first tool call is GetTime.
</think>

<tool_call return="one">
    {
        "0": {
            "GetTime": {}
        },
        "1": {
            "GetCurrentLocation": {}
        },
        "2": {
            "FindRestaurantsByLocation": {
                "location": "API_RESPONSE_1"
            }
        },
        "3": {
            "FilterByCuisine": {
                "cuisine": "French",
                "restaurants": "API_RESPONSE_2"
            }
        },
        "4": {
            "FilterByOpeningHours": {
                "time": "API_RESPONSE_0",
                "restaurants": "API_RESPONSE_3"
            }
        }
    }
</tool_call>
```

E.2 Why trajectory supervision fails

The following cases are taken verbatim from the ToolRL rollout logs. They show that scoring against a single annotated reference mis-scores composition in both directions.

Case 1: a valid alternative is penalized.

- **Reference:** `{GetLocation(), GetDate(), GetAirQuality(date=R1, location=R0)}`
- **Generated:** `{GetCurrentLocation(), GetDate(), GetAirQuality(date=R1, location=R0)}`

The model emits `GetCurrentLocation()` in place of `GetLocation()`: a different API that returns the same result and yields a correct final answer. ToolRL assigns 2.5 on $[-3, 4]$; TIER assigns 1.0 on $[0, 1]$.

Case 2: a broken chain is not penalized.

- **Reference:** `{GetLocation(), GetRestaurantsByLocation(location=R0)}`
- **Generated:** `{GetLocation(), GetRestaurantsByLocation(location='San Francisco')}`

The model calls `GetLocation()` correctly, then hallucinates `location='San Francisco'` instead of binding `API_RESPONSE_0`. The true location is New York, so the returned list is wrong. Because trajectory matching never executes the chain, this is invisible; only the literal parameter mismatch is caught. ToolRL assigns 2.5; TIER assigns 0.5.

A correct chain and a broken chain therefore receive identical reward. This is not a tuning artifact but a consequence of scoring against a single reference without execution, and it is why we attribute ToolRL's training instability (Figure 1) to the reward rather than to the optimizer.

F Tool-Call Efficiency

Solved queries. The grader does not penalize extra calls: a response receives full credit if every call is valid and the final answer is correct, so a solution may include superfluous calls that leave the returned output unchanged. Nevertheless, every solved query at depth d contains exactly d calls, with a standard deviation of 0.000 across all depths and methods. No method, TIER included, obtains its accuracy by issuing additional calls.

Failed attempts. The informative comparison is over parseable-but-incorrect attempts, reported in Table 7.

Table 7: Average number of tool calls emitted on parseable-but-incorrect attempts, by required depth. Values are mean $\pm$ 95% Student-t CI over $N = 10$ evaluation runs of a single trained model.

	Required depth						
Method	0	1	2	3	4	5	6
TIER	1.35 $\pm$0.24	1.00 $\pm$0.00	2.62 $\pm$0.42	2.35 $\pm$0.37	3.77 $\pm$0.36	3.00 $\pm$1.71	3.57 $\pm$1.05
Simple	1.70 $\pm$0.56	0.98 $\pm$0.02	1.02 $\pm$0.00	1.06 $\pm$0.01	1.44 $\pm$0.06	1.09 $\pm$0.07	1.08 $\pm$0.06
ToolRL	2.10 $\pm$0.07	1.06 $\pm$0.02	2.01 $\pm$0.00	2.95 $\pm$0.02	3.82 $\pm$0.04	4.08 $\pm$0.07	4.75 $\pm$0.07

Three patterns emerge. First, TIER's failures are predominantly truncations rather than over-calls: at depths 3–6 its incorrect attempts emit fewer calls than the query requires. Second, Simple collapses to approximately one call at every depth ≥ 2, indicating that it does not attempt composition at all, which accounts for its near-zero accuracy in those bins. Third, ToolRL's failed attempts average 4.08 and 4.75 calls when five and six are required, so they fall short of the required chain length, an effective chain-length ceiling; below the ceiling its accuracy tracks how constrained each step is, while above it the required chain cannot be produced at all. We attribute this chain-length ceiling to ToolRL's penalizing of valid alternatives, which results in unstable policy updates. The zeros at depths 5–6 are therefore a capability boundary rather than a point on a difficulty gradient.

TIER over-calls in two settings: at depth 2 (2.62 calls on failed attempts) and at depth 0 (1.35). The depth-0 case is the abstention failure mode discussed in Section 4.2: depth-0 items require recognizing that no tool applies, and the characteristic error is invoking a plausible but inapplicable tool.

G Extending TIER beyond DepthBench

G.1 What TIER requires of a dataset

TIER computes all four reward components from function schemas and runtime execution; no reference trajectory is consulted when a real backend is available. What the method requires is therefore an executable tool environment, not DepthBench specifically. Source datasets fall into three categories:

1. **Executable backend; no adaptation.** Tool descriptions are supplied in conventional schema format, and our reward computation and parser apply unchanged. Any corpus that can execute its own tools is immediately compatible.
2. **Reference trajectories but no backend; requires programmatic surrogate.** An executable variant is constructed mechanically from the released schemas, as described in Section G.3. No manual annotation is involved. Because no real return values exist, the answer check falls back on the released trajectory.
3. **Neither; outside scope**, as for any execution-grounded reward.

ToolACE and xLAM fall into the second category. We adapt and train on both below, both to test whether DepthBench's compact, composition-focused corpus is competitive with established large-scale datasets and to characterize what the adaptation costs.

G.2 Adapting existing tool-use datasets for TIER

DepthBench is deliberately compact: 1.7K examples over 163 tools, compared to $\sim$10K examples over $\sim$26K tools in ToolACE and $\sim$60K examples over $\sim$4K tools in xLAM. We train Qwen3-8B under TIER on adapted versions of both, using the same reward formulation, optimizer, and rollout budget as in Section 4.

G.3 Adaptation procedure

ToolACE and xLAM are released as supervised fine-tuning corpora providing a single annotated trajectory per query, without an executable backend; neither natively supports execution-grounded rewards. We construct executable variants by parsing each function schema to recover its parameter set, declared types, and a required/optional partition. Because the two corpora encode this partition differently, we apply dataset-specific rules. ToolACE exposes an explicit `required` field alongside per-parameter defaults; we mark a parameter as required iff it appears in `required`, with the field taking precedence when a parameter both appears there and declares a default. xLAM exposes no `required` field, so we treat any parameter with a declared default as optional and all others as required. This partition recovery is the only dataset-specific component of the procedure; the remainder is mechanical.

A call passes parsing when every supplied argument belongs to the schema, all values type-check against their declarations, and every required parameter is present; optional parameters may be omitted. Parsed calls return a deterministic response that echoes the resolved arguments, with omitted optionals materialized to their declared defaults prior to echoing. Because the source corpora contain no golden tool outputs, this design isolates correctness at the parameter-passing layer; an omitted optional and the same optional passed at its default value yield identical responses by construction, so optional-parameter choices cannot influence downstream correctness through the execution layer.

On these corpora, the answer reward R_{answer} is computed by a relaxed match against the released trajectory: required parameters must match the gold values exactly, while optional parameters are not scored. The criterion is strictly stronger than parsing success and strictly weaker than full trajectory string match—the latter would penalize superficial reordering or default-value rewrites that leave behavior unchanged.

G.4 Scope

ToolACE contains multi-turn splits in which composition spans across turns. The present work focuses on single-turn composition, so these splits are excluded from both training and evaluation.

Table 8: BFCL v3 performance for Qwen3-8B trained under identical TIER supervision on each corpus.

DATASET (SIZE)	OVERALL	NON-LIVE AST	LIVE	MULTITURN	RELEVANCE	IRRELEVANCE
DEPTHBENCH (1240)	**68.92**	85.94	81.96	**39.12**	**77.78**	87.65
TOOLACE (10K)	64.66	85.90	**82.81**	24.12	61.11	**91.01**
XLAM (60K)	63.02	**89.29**	79.16	21.22	**77.78**	83.92

G.5 Results

Table 8 reports BFCL v3 performance for models trained on each corpus under identical TIER supervision. Despite being roughly an order of magnitude smaller, the DepthBench-trained model achieves the strongest overall performance. The gap is concentrated on the Multiturn metric (39.1 vs. 24.1 and 21.2); single-turn metrics (Non-Live AST, Live) lie within a few points across the three corpora and do not separate them.

G.6 Analysis

Two dataset-level properties appear to drive the gap. First, xLAM contains frequent mismatches between released trajectories and the corresponding tool schemas — argument names, types, or values in the annotated trajectory do not match the declared signature. Under execution-grounded rewards, these inconsistencies penalize schema-conformant calls, because the schema and the trajectory disagree on what counts as correct. Second, ToolACE pairs a very large tool vocabulary ($\sim$26K) with relatively few examples ($\sim$10K), so most tools appear in at most one query. The resulting models invoke individual tools too rigidly to handle the argument variation present at evaluation time. Neither property is visible from single-call accuracy, which is why aggregate function-calling benchmarks fail to expose the gap that BFCL Multiturn surfaces.

G.7 What the adaptation costs

Beyond these corpus-specific properties, a structural limitation applies to both adapted datasets and bounds what the comparison can show. These results should not be read as an upper bound on TIER's transferability, because the adaptation is lossy in an identifiable way. Since the source corpora contain no golden tool outputs, a surrogate can only echo its resolved arguments. Schema adherence and argument binding remain fully observable, but execution success is degenerate — a well-formed call always succeeds — and answer correctness can only be checked by matching required parameters against the released trajectory. Two of TIER's four components therefore lose their execution grounding on these corpora: the execution reward carries no signal, and the answer reward reverts to reference matching, the very supervision TIER is designed to avoid. The model trains against a signal that rewards matching the reference without verifying that the calls accomplish anything. Composition suffers most, since the failure mode a chain must avoid is precisely a well-formed call whose output is wrong.

This is a property of the source data rather than of the adaptation procedure, and it cannot be repaired by better parsing. A corpus supplying a single annotated trajectory without an environment can certify that a call *matches* the reference; it cannot certify that the call *works*. Scale does not substitute: adding annotated trajectories does not approach execution grounding asymptotically, which is why a corpus a fiftieth the size of xLAM's outperforms it here, and why we hand-authored DepthBench with executable backends (Appendix A).

H Loss Aggregation Ablation

To verify that the gains reported in this paper are driven by reward design rather than the choice of optimization algorithm, we train three models using different loss aggregations—standard GRPO, batch-normalized GRPO, and DAPO-style token-level aggregation [Yu et al., 2025] (DAPO below)—under identical reward formulation, model (Qwen3-8B), rollout budget, and hyperparameters. The only variable across conditions is the loss aggregation. Figure 4 shows training reward curves as exponentially weighted moving averages of the per-step batch-mean reward (smoothing constant

Table 9: Loss aggregation ablation on DepthBench. All configurations use the same TIER reward, model (Qwen3-8B), and rollout budget. Only the loss aggregation differs. Results are from a single run per configuration.

Loss aggregation	Overall Acc	0-Step	1-Step	2-Step	3-Step	4-Step	5-Step	6-Step
Standard GRPO	97.13	98.04	97.92	94.74	96.00	**97.78**	90.00	**90.00**
Batch-Normalized GRPO	**98.56**	98.04	**99.38**	97.89	**100.00**	93.33	90.00	**90.00**
DAPO	93.21	**100.00**	94.79	**98.95**	76.00	88.89	**100.00**	70.00

$\alpha = 0.01$); shaded regions are ± 1 standard deviation of the unsmoothed reward over a 50-step rolling window.

H.1 Training Dynamics

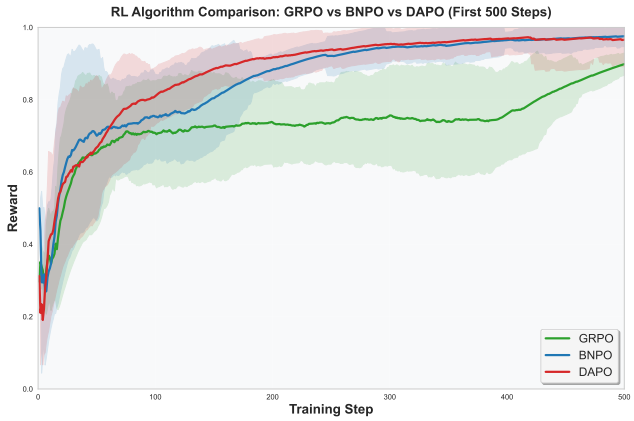

Figure 4: Training reward over the first 500 steps for three loss aggregations (Standard GRPO, Batch-Normalized GRPO, DAPO) under identical fine-grained, trajectory-invariant reward formulation. Shaded regions show ± 1 standard deviation. DAPO converges fastest, Batch-Normalized GRPO converges steadily to the highest level, and GRPO exhibits a delayed plateau-then-jump pattern. In the legend, BNPO denotes Batch-Normalized GRPO.

DAPO converges fastest, reaching approximately 0.9 reward by step 100 and stabilizing with moderate variance. Batch-Normalized GRPO converges more gradually, reaching near-ceiling reward by step 200 and continuing to improve through step 500. Standard GRPO exhibits a distinct trajectory: it plateaus around 0.72–0.75 for an extended period (steps 100–350), followed by a delayed but sharp improvement after step 400, eventually approaching 0.9. This plateau-then-jump behavior suggests that GRPO requires a longer exploration phase before discovering effective compositional strategies, likely due to its reliance on group-relative comparisons. GRPO also shows higher variance throughout training, indicating less stable optimization compared to Batch-Normalized GRPO and DAPO.

H.2 Evaluation Results

We report accuracy on the composition benchmark stratified by depth in Table 9.

All three variants achieve over 90% overall accuracy, confirming that the fine-grained TIER reward design is the primary driver of compositional capability rather than any specific loss aggregation.

However, the variants differ in their depth profiles. Batch-Normalized GRPO achieves the highest overall accuracy (98.56%), with every depth between 90% and 100%. Standard GRPO achieves comparable overall accuracy (97.13%) and the narrowest range across depths (90.00–98.04%), despite slower convergence during training. DAPO exhibits the greatest variance across depths: while it achieves 100% at depths 0 and 5, it drops to 76.00% at 3-step and 70.00% at 6-step.

DAPO converges fastest during training; committing to a policy early may leave some compositional patterns under-covered. In contrast, GRPO's extended exploration phase appears to improve coverage across compositional patterns, leading to more uniform depth performance.

Based on these results, we adopt Batch-Normalized GRPO as the default loss aggregation due to its highest overall accuracy. The key finding, however, is that *all three variants enable multi-step composition when paired with fine-grained, trajectory-invariant, execution rewards*. Whereas trajectory-supervised rewards yield near-zero accuracy at depths 5–6, TIER yields high multi-step accuracy under all three variants, with GRPO and Batch-Normalized GRPO achieving consistently high accuracy and DAPO showing more variable but still substantial gains.

I SFT Baseline

The supervised fine-tuning (SFT) baseline in Table 3 is trained on the same 1,240-example split as the RL arms. Because DepthBench provides one gold tool-call trajectory per query but no reasoning traces, we construct the SFT targets as follows. We first train Qwen3-8B with TIER and use the trained model to generate a rollout for each training query, consisting of its reasoning trace and its tool-call trajectory. We execute every generated trajectory and keep it only if it produces the expected output; since the TIER-trained model is already highly accurate, this covers most queries. For the remaining queries, we write the reasoning trace and tool-call trajectory by hand. We then fine-tune Qwen3-8B on the resulting trajectories with a standard supervised objective. The SFT dataset is included in our released data.

J Training Compute

All training runs were conducted on an internal cluster using 2× NVIDIA A100 (80GB) GPUs per run. Each run trained for 1,000 optimization steps and required approximately 20 wall-clock hours (≈40 GPU-hours per run). In total, we trained roughly 100 model configurations, corresponding to approximately 4,000 GPU-hours of training compute. Each trained model was subsequently evaluated on BFCL and DepthBench using a single A100 (80GB) GPU, amounting to approximately 1,680 GPU-hours of evaluation compute. The full research project additionally incurred approximately 1,500 USD in OpenAI API costs across dataset evaluation and exploratory evaluation on ComplexFuncBench that we ultimately excluded from the main results (see Section 7). Beyond the runs reported above, additional GPU compute was spent on preliminary and discarded experiments that are not included in this paper.

K Extending TIER to the multi-turn setting

Our evaluation protocol is single-turn: the model emits the entire composition in one shot, with no execution feedback and no opportunity to revise. Extending TIER to turn-budgeted settings requires three changes. First, a stateful environment — tool-state persistence and a user or replay simulator on top of the executable backend. Second, credit assignment across turns, aggregating per-turn rewards into a trajectory return. Third, and most substantively, rewarding recovery: the reward must credit successful correction of an observed failure, since a naive per-turn penalty would train toward risk-averse under-calling. We note that TIER already improves on BFCL v3 Multiturn without any of these (Section 4.4), which suggests the signal transfers even in its current form.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The claims made in the abstract and introduction match the experimental results in Section 4

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Limitations of the work are discussed in Section 7

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

Justification: This work does not discuss any theoretical results. All empirical results are explained along with the experiment setup.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: This work discusses the TIER framework in detail for reproducibility. Training data and code is publicly released (refer Section 1). System prompt, dataset construction and adaptation procedure, training specification, hardware related specifications can be found in Appendix A, C, D, and J

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: Data and code are publicly released. See Section 1

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Experiment setup and training environment are discussed in Section 3, and Section 4

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: Tables 1 and 7 report the mean $\pm$ 95% Student-t confidence interval over 10 evaluation runs of a single trained model. These intervals capture variation from evaluation sampling, not from training seeds; repeated RL training runs were computationally prohibitive. All other results are single runs, as stated in their table captions.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.

- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: We discuss relevant details in Appendix J.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: Yes

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: Impact is discussed in Section 8

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.

- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: No such risks.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: Our work uses several third-party models, datasets, and software components, each governed by their respective licenses. Qwen3-8B is used under the Apache 2.0 License. The BFCL v3 benchmark is used under the Apache 2.0 License. Training and infrastructure rely on open-source libraries including TRL (Apache 2.0), DeepSpeed (MIT License), and PyTorch (BSD-style License), along with standard scientific Python packages such as NumPy and pandas (BSD-style licenses). NestFUL and ToolACE are also under the Apache 2.0 License and xLAM is released under the Creative Commons Attribution 4.0 License.

    All newly created assets in this work, including DepthBench, training data, and model checkpoints, are released under the CC-BY 4.0 license. We comply with all applicable third-party licenses and redistribute only artifacts permitted under their respective terms.

    Guidelines:

- The answer [N/A] means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: All assets under license CC-BY 4.0. Assets are publicly released.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: Does not apply

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: Does not apply

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [N/A]

    Justification: LLMs were only used for writing, editing, or formatting purposes.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.