

FunReason-MT Technical Report: Overcoming the Complexity Barrier in Multi-Turn Function Calling

Zengzhuang Xu^{1†*}, Bingguang Hao^{1†*}, Zechuan Wang^{2*}, Yuntao Wen^{1*}, Maolin Wang^{3*}, Yang Liu^{2*}, Long Chen^{1*}, Dong Wang^{1*}, Yicheng Chen^{1*}, Cunyin Peng^{1*}, Chenyi Zhuang^{1‡}, Jinjie Gu¹, Leilei Gan², Xiangyu Zhao³, Shi Gu^{2‡}

 ¹AWorld Team, Inclusion AI  ²Zhejiang University  ³City University of Hong Kong

 FunReason-MT Dataset

 FunReason-MT Model

 Project FunReason-MT

Abstract

Function calling (FC) empowers large language models (LLMs) and autonomous agents to interface with external tools, a critical capability for solving complex, real-world problems. As this ability becomes increasingly central to advanced AI systems, the need for high-quality, multi-turn training data to develop and refine it cannot be overstated. Existing data synthesis methods, such as random environment sampling or multi-agent role-playing, are not powerful enough to generate high-quality data in real-world environments. Practical challenges come in three folds: targeted model training, isolation of tool architecture, and multi-turn logical dependency. To address these structural deficiencies, we present FunReason-MT, a novel data synthesis framework for real-world multi-turn tool use. FunReason-MT resolves the complexity barrier in multi-turn FC data by employing 1) *Environment-API Graph Interactions* to gather varied high-quality trajectories, 2) *Advanced Tool-Query Synthesis* to simplify hard query construction, and 3) *Guided Iterative Chain* for sophisticated CoT generation. Evaluations on Berkeley Function-Calling Leaderboard (BFCLv3) demonstrate the power of our framework: a 4B model built upon FunReason-MT generated data achieves state-of-the-art performance among comparable-sized models, outperforming most close-source models. Further performance improvements on BFCLv4 confirm that FunReason-MT provides a reliable and robust source for agentic learning.

This is part of the open-source Project  ¹AWorld, Inclusion AI. The training data and model weights are released.

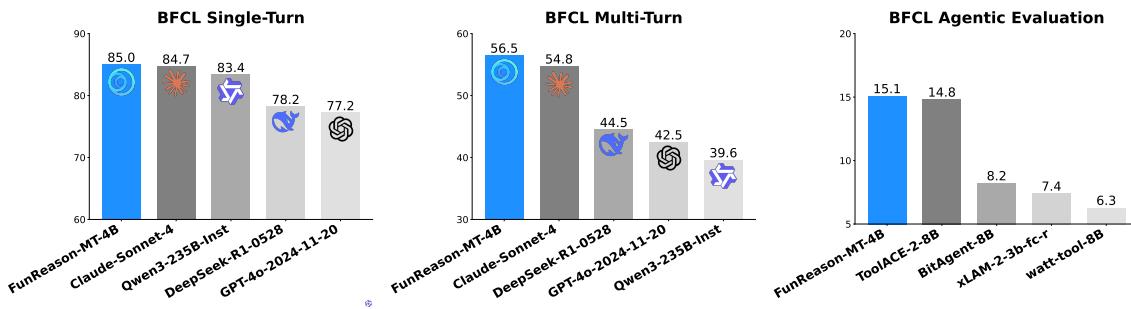


Figure 1: Performance on BFCL Single-Turn, Multi-Turn and Agentic Evaluation.

^{*}Equal Contribution. The two authors designed the system.

[†]Core Contributor. These authors participated in algorithm development, data engineering, and infrastructure.

[‡]Corresponding Author. chenyi.zcy@antgroup.com, gus@zju.edu.cn

Work was done during Bingguang's internship at Ant Group. bingguanghao7@gmail.com



Figure 2: We summarize the core challenges in multi-turn function-calling data generation into three aspects from left to right: Targeted Model Training, Isolation of Tool Architecture, Multi-Turn Logical Dependency.

1 Introduction

Large Language Models (LLMs) are profoundly transforming human-computer interaction, largely through Function Calling (FC), the pivotal technology enabling LLMs to interface with external tools [WANG et al. \(2025\)](#). Although contemporary foundation models possess inherent tool-use capability, unlocking more advanced applications in this domain is hindered by a critical bottleneck: The construction of effective, high-quality data, particularly for multi-turn trajectories. Existing data generation research predominantly relies on constrained methodologies, such as random sampling within environments or Multi-Agent System (MAS) role-playing simulations [Huang et al. \(2025\)](#); [Prabhakar et al. \(2025\)](#); [Liu et al. \(2024\)](#); [Zeng et al. \(2025\)](#). Crucially, random sampling fails to capture rare, complex events, while MAS role-playing often defaults to simplistic ‘happy-path’ scenarios or suffers from a lack of true diversity [Han et al. \(2024\)](#). Thus these approaches inherently produce data that is limited in both effectiveness and diversity, resulting in a substantial training performance ceiling.

As illustrated in Figure 2, we categorize the challenge in multi-turn FC data generation into three critical, interconnected issues that diminish data quality and model stability. **1) Targeted Model Training.** Existing random sampling techniques fail to controllably and targetedly construct model training scenarios requiring the collaborative use of a target complex tool with others across diverse, multi-turn trajectories. **2) Isolation of Tool Architecture.** Due to the modular inputs of individual tools rather than holistic integration, direct environmental sampling and MAS role-playing struggle to synthesize hard, logical-jump queries. **3) Multi-Turn Logical Dependency.** In complex multi-turn dialogues, each step of the Chain-of-Thought (CoT) is a validation and reconstruction of the reliance on prior logic. Current Reasoning Large Language Models (RLLMs) often falter when generating CoT in unexplored environments, preventing the acquisition of complete and accurate multi-turn trajectories.

To fundamentally resolve these structural limitations, we introduce the **Function Call Reasoning Multi-Turn** (FunReason-MT) data synthesis framework. Our framework prioritizes robustness and efficiency, and incorporates three specific core components to tackle the outlined challenges.

- **Environment-API Graph Interactions.** We establish an API graph based on interdependencies between tools and environments, enabling the collection of multi-turn trajectories centered around a target complex API within varied scenarios, ultimately training models for targeted mastery and collaborative tool use.
- **Advanced Tool-Query Synthesis.** We use the tool-state pairs collected during Environment-API Graph Interactions to construct an advanced tool that eliminates the need to input the modularized sub-tools when queries are generated in reverse. This inherently makes the execution of prerequisite tools implicit.
- **Guided Iterative Chain.** We input the hard query, the set of sub-tools, and the advanced tool description to obtain CoT. We then implement an iterative feedback loop: The analysis identifying the specific reasons for any failure to match the ground truth is continuously integrated as a corrective hint until the CoT successfully leads to the right FC or meets the maximum attempts.

We validate the FunReason-MT framework on the challenging Berkeley Function-Calling Leaderboard (BFCLv3), encompassing both single-turn and multi-turn tests [Patil et al.](#). Experiments demonstrate that

our data construction method consistently outperforms approaches based on MAS role-playing or random environmental sampling, and achieves state-of-the-art performance among comparable-sized models. This success affirms that FunReason-MT effectively overcomes the complexity barrier in multi-turn FC data generation, establishing an essential knowledge supplement for future LLM environment exploration. We also evaluate our models on BFCLv4, an out-of-distribution (OOD) benchmark, to further assess their agentic ability. Although our data is unrelated to the tasks evaluated by BFCLv4 (Web Search and Memory), our models still show improvement in these specific areas. We believe our study is well-positioned to lay a robust and enduring foundation that facilitates the future progression of agentic learning [Zhang et al. \(2025\)](#); [Ke et al. \(2025\)](#).

2 Motivation

Our motivation stems from the insight that the identified "performance ceiling" is not merely a data quantity issue, but a fundamental failure of the existing data generation paradigm. We recognized that "bottom-up" approaches like random sampling and MAS role-playing are structurally incapable of producing the high-complexity, long-tail trajectories needed for true mastery. These methods merely hope for complexity to emerge from simple interactions, suffer from low controllability and poor reliability.

This drove us to devise a new, "top-down" construction methodology that can explicitly direct the model to master a specific complex tool within a targeted complex scenario. Our approach is motivated by the "why" and "how" of this shift: instead of hoping for targeted complex tool involved trajectories, we engineer them by modeling the environment's structural dependencies (the API graph). Instead of sampling simple tool combinations, we guarantee complex, logical-jump queries by generating them in reverse. And finally, instead of accepting flawed reasoning, we enforce CoT accuracy through a guided feedback loop, thus directly targeting the three core challenges that plague existing methods and corresponding to solving the issues of **reliability** and **complexity** respectively.

3 Notation

In order to describe the generation process of the FunReason-MT framework, we introduce the key symbols and concepts used to model the system's components and operations. The core concepts and models that manage the system are defined in Table 1.

Symbol	Concept
\mathcal{S}	Multi-Environment Simulation Space.
\mathcal{T}	Tool Set.
$\mathcal{G} = (\mathcal{T}, \mathcal{R}, \mathcal{P})$	API Relation Graph (Tools, Dependencies \mathcal{R} , Parameters \mathcal{P}).
A_T, A_Q, A_R, A_C	LLM Agents (Tooling, Querying, Reasoning, Critiquing).
$T_a \in \mathcal{T}$	Target Tool.
M	Number of tool calls per round.
N	Number of trajectory rounds.
K_{\max}	Maximum self-correction attempts.
Traj	Multi-turn trajectory.
Turn _i	i-th single-turn content.

Table 1: Core Concepts and Models of FunReason-MT.

4 Methodology

As shown in Figure 3, our proposed FunReason-MT framework comprises three core phases: (1) **Phase I: Environment-API Graph Interactions** constructs a valid multi-step execution trace by sampling tool calls from the API Relation Graph (§4.1); (2) **Phase II: Advanced Tool-Query Synthesis** reverse-engineers a hard-to-solve data sample and synthesizing a corresponding challenging Hard Query (Q_{hard}) (§4.2); (3) **Phase III: Guided Iterative Chain** utilizes an iterative, self-correction loop to refine the CoT and ensure its logical consistency with the ground truth answer (§4.3).

4.1 Phase I: Environment-API Graph Interactions

A central challenge in generating multi-turn tool-use trajectories is to devise a sampling strategy that is both **executable** and **purposefully directed**. To achieve this, FunReason-MT framework is designed to satisfy two

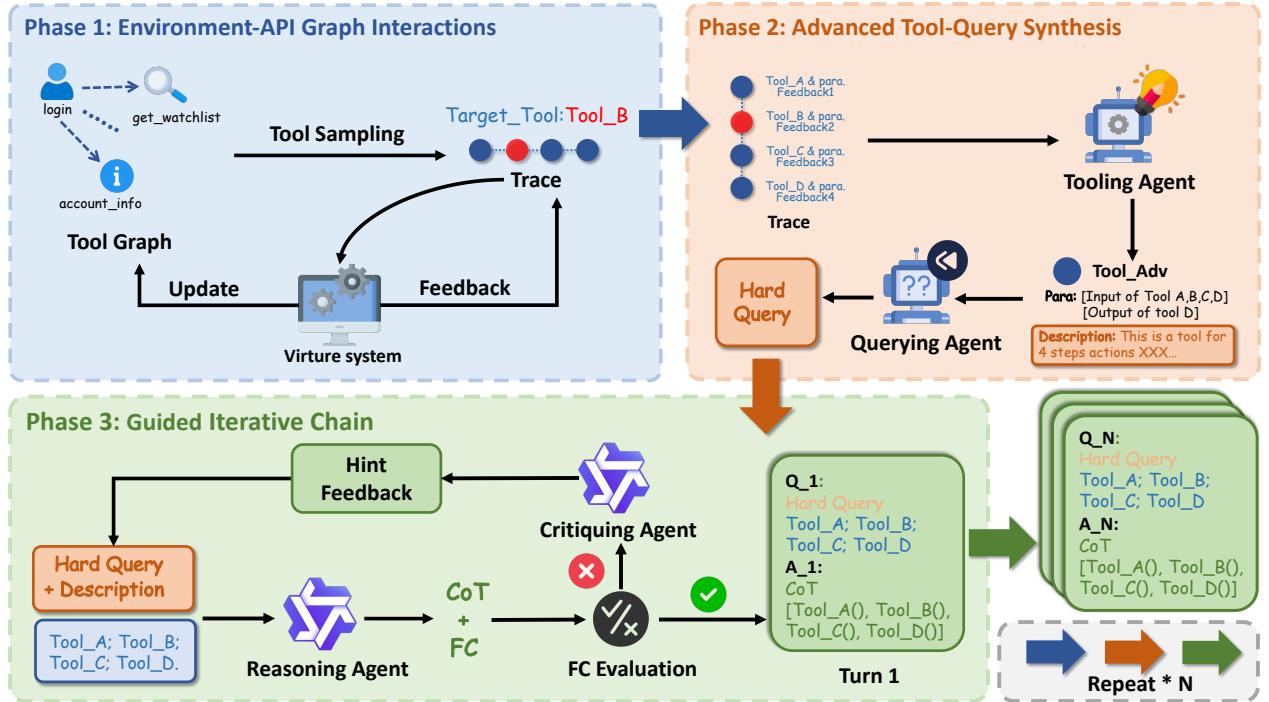


Figure 3: Three Phases of FunReason-MT.

critical constraints simultaneously: *execution correctness* (i.e., respecting tool dependencies) and *goal-directed efficiency* (i.e., prioritizing T_a). We detail these two components below.

Formalizing Tool Call Legality

To ensure a valid execution sequence, we introduce a strict dependency constraint: Tool T_i is only callable if all its necessary preconditions, $\text{Prerequisite}(T_i)$, are satisfied by the set of already executed tools, $\mathcal{T}_{\text{called}}$. We formalize this essential constraint using the tool legality check I :

$$I(T_i, \mathcal{T}_{\text{called}}) = \mathbf{1}_{\{\text{Prerequisite}(T_i) \subseteq \mathcal{T}_{\text{called}}\}}$$

where $\mathbf{1}_{\{\cdot\}}$ is the indicator function, returning 1 if the condition is true and 0 otherwise.

Tool Sampling with Priority

While random sampling over the tool graph can satisfy execution correctness, it often fails to reach the target tool T_a efficiently. To address this, we introduce a Directed Sampler, denoted as **SampleTool**, which explicitly biases the sampling process toward T_a . Specifically, **SampleTool** adopts a greedy heuristic strategy that operates over the set of legal tools $\mathcal{T}_{\text{legal}}$ and prioritizes those minimizing the graph distance $\text{dist}(\cdot, \cdot)$ to T_a . The sampling policy of **SampleTool** is formally defined as follows:

$$T_s = \text{SampleTool}(T_a, \mathcal{T}_{\text{called}}) = \begin{cases} \text{rand}(\mathcal{T}_{\text{legal}}) & \text{if } T_a \in \mathcal{T}_{\text{called}} \\ T_a & \text{if } I(T_a, \mathcal{T}_{\text{called}}) = 1 \wedge T_a \notin \mathcal{T}_{\text{called}} \\ \underset{T_k}{\operatorname{argmin}} \{\text{dist}(T_k, T_a)\} & \text{otherwise} \end{cases} \quad (1)$$

The chosen tool T_s is executed using sampled parameters P_s as the call $C_s = (T_s, P_s)$. This action yields environmental feedback E_i , triggers an update to the system state S_j , and consequently updates the API Graph \mathcal{G} , legal tools $\mathcal{T}_{\text{legal}}$ and executed tools $\mathcal{T}_{\text{called}}$. The resultant single-turn execution sequence is recorded as the trace: $\text{Turn}_i = (C_1, E_1, C_2, E_2, \dots, C_M, E_M)$.

4.2 Phase II: Advanced Tool-Query Synthesis

While the execution trace Turn_i ensures correct tool execution, it does not directly capture the abstract intent behind those operations. To address this, Phase II reverses the process by synthesizing a challenging query

Q_{hard} that compels the model to reason over the entire trace as a single high-level abstraction.

1. **Advanced Tool Generation.** Given the execution trace Turn_i , the *Tooling Agent* (A_T) synthesizes a single composite abstraction, denoted as the advanced tool T_{adv} :

$$T_{\text{adv}} = A_T(\text{Turn}_i) \quad (2)$$

where A_T abstracts the multi-step execution trace into a unified high-level operation that encapsulates the functionality and interdependencies of all sub-tools within Turn_i .

2. **Hard Query Construction.** Conditioned on the synthesized advanced tool T_{adv} , the *Querying Agent* (A_Q) generates a challenging hard query Q_{hard} that explicitly requires the use of T_{adv} for resolution. Formally, the construction process is expressed as:

$$Q_{\text{hard}} = A_Q(T_{\text{adv}}, \epsilon) \quad (3)$$

where ϵ denotes a synthesis noise that increases task difficulty and encourages generalization.

4.3 Phase III: Guided Iterative Chain

While the construction of Q_{hard} promotes higher-level reasoning, it also increases the risk of logical inconsistency or incomplete Chain-of-Thought (CoT) generation. To address this, Phase III introduces the Guided Iterative Chain, a feedback-driven refinement process that iteratively improves CoT quality and coherence until the reasoning converges to a reliable solution or reaches the maximum attempts.

Initial Reasoning Attempt and Correction Loop

The Reasoning Agent A_R begins by attempting to resolve the hard query, Q_{hard} , utilizing both the advanced tool description $\text{Des}(T_{\text{adv}})$ and its primitive components \mathcal{T}_{sub} . This first attempt yields the initial output $O^{(0)} = \{\text{CoT}, \text{FC}\}$, comprising a CoT and a final Function Call.

The generated function call ($\text{FC}^{(k)}$) is subsequently validated against the known ground truth G ($G = (C_1, C_2, \dots, C_M)$) using a verification function $\text{Validate} : \text{FC} \times G \rightarrow \{\text{Pass}, \text{Fail}\}$.

1. **Error Analysis (Upon Failure):** If the current attempt fails, the *Critiquing Agent* (A_C) analyzes the incorrect function call $\text{FC}^{(k)}$ with respect to the ground truth G , identifies the specific error, and generates targeted corrective feedback: $\text{Error}^{(k)} = A_C(\text{FC}^{(k)}, G)$.
2. **Prompt Augmentation:** The corrective error feedback is then integrated into the *Reasoning Agent's* context as $\text{Prompt}^{(k+1)}$. This addition implicitly creates a feedback loop, effectively guiding the subsequent attempt.
3. **Self-Correction:** The *Reasoning Agent* A_R is re-invoked with the augmented prompt to generate a refined solution, $O^{(k+1)} = A_R(\dots, \text{Prompt}^{(k+1)})$.

The maximum allowed limit for this iterative correction process is K_{max} attempts. A data sample is only retained if the validation function $\text{Validate}(\text{FC}^{(k)}, G)$ returns Pass before the iteration cap is reached.

4.4 Scaling from Single-Turn Trace to Multi-Turn Trajectory

The full pipeline is iterated N times to scale the single-turn generation into a comprehensive, high-quality multi-turn trajectory, Traj .

The overall data generation process starts by initializing Traj as an empty set. The framework then sequentially executes the three phases for each of the N required data rounds:

Step 1: Environment API-Graph Interactions: A target tool T_a is chosen from the primitive tool set \mathcal{T} . This choice dictates the generation of a complete execution trace Turn_1 within the simulation environment, following the procedure outlined in Phase 1.

Step 2: Advanced Tool-Query Synthesis: The successful trace is then given to the LLM agents A_T and A_Q . These agents collaborate to synthesize the Advanced Tool (T_{adv}) and its corresponding Hard Query (Q_{hard}), thereby forming the core of the new data sample.

Step 3: Guided Iterative Chain: Immediately following synthesis, the data sample undergoes the crucial Guided Iterative Chain verification process. This step employs the reasoning and critiquing agents (A_R, A_C) to attempt solving the hard query and validate results against the ground truth answer G .

To ensure the resulting multi-turn trajectory is reliable and complex, the entire process detailed above will be repeated N times.

5 Experiments

5.1 Experimental Setup

Training Details. Considering the length of multi-turn conversation and CoT, we use Qwen3-4B-Instruct-2507 with 256K long-context understanding as our base model Yang et al. (2025). The collected trajectories are split at each assistant response and the model is only trained on the CoT and answer at current turn. We collect 10000 multi-turn samples, and to enhance the diversity of the dataset, we also jointly train model with function calling data from APIGen. We use LLama-Factory Zheng et al. (2024) and Verl Sheng et al. (2024) library to train models with implementations from Hao et al. (2025b) and Hao et al. (2025a).

Benchmarks. Our primary evaluation is performed on the BFCLv3, which is designed to assess the model’s performance on both Single-Turn and Multi-Turn tool-calling tasks. To further evaluate the data influence on the model’s agentic capability, we extend our assessment to the **Held-out (OOD)** BFCLv4 benchmark. BFCLv4 is engineered for the evaluation of agentic performance with Web Search and Memory. We present all the results in terms of accuracy (%).

5.2 Main Results

Model	Multi-Turn					Single-Turn		
	Overall	Base	Miss Func	Miss Param	Long Context	Overall	Non-Live	Live
Close Source Model								
GPT-5-2025-08-07	28.50	33.50	29.50	23.00	28.00	65.59	72.92	58.25
GPT-4o-2024-11-20	42.50	55.50	34.50	29.00	51.00	77.21	83.88	70.54
Claude-Sonnet-4-20250514	54.75	<u>64.00</u>	54.00	47.50	<u>53.50</u>	<u>84.72</u>	<u>88.38</u>	<u>81.05</u>
Gemini-2.5-Pro	25.00	25.50	26.00	24.50	24.00	74.50	85.04	63.95
o3-2025-04-16	38.38	44.00	40.50	31.50	37.50	53.01	39.98	66.03
Grok-4-0709	36.12	44.00	31.00	26.00	43.50	79.80	85.21	74.39
Open Source Model								
Moonshotai-Kimi-K2-Inst	41.25	51.00	43.00	31.00	40.00	80.80	84.02	77.57
DeepSeek-R1-0528	44.50	54.50	41.00	36.50	46.00	78.22	75.73	80.90
Qwen3-235B-A22B-Inst-2507	39.62	53.50	34.50	27.50	43.00	83.37	90.12	76.61
Llama-4-Maverick	17.88	23.50	18.00	14.00	16.00	80.90	88.15	73.65
ToolACE-2-8B	37.00	47.00	31.00	28.00	42.00	82.54	87.87	77.20
BitAgent-8B	37.75	46.50	37.50	24.00	43.00	81.71	87.33	76.09
watt-tool-8B	37.88	45.50	39.00	24.00	43.00	81.71	87.54	75.87
xLAM-2-3b-fc-r	57.12	73.50	<u>55.00</u>	54.50	45.50	73.82	83.31	64.32
ToolACE-MT	40.25	57.50	31.50	34.00	38.00	78.23	84.94	71.52
Qwen3-4B-Inst-2507	15.75	19.00	15.50	12.50	16.00	78.19	86.35	70.02
+FunReason-MT (SFT)	46.90 _{+31.15}	53.20	47.10	40.40	46.90	81.97 _{+3.78}	83.36	80.57
+FunReason-MT (RL)	56.50 _{+40.75}	61.50	60.00	48.00	<u>56.50</u>	<u>85.02_{+6.83}</u>	88.38	81.65

Table 2: Performance on BFCLv3 (last updated August 26, 2025). The best result within each category is highlighted in **bold**. The second best result is underlined.

As shown in Table 2, on the BFCLv3 benchmark, FunReason-MT yields notable improvements on Qwen3-4B-Instruct, raising the Multi-Turn score from 15.75 to 46.90 (+31.15) after SFT and to 56.50 (+40.75) after RL. Despite its 4B size, the FunReason-MT RL-trained model surpasses both strong open-source models (e.g., Kimi-K2-Inst, DeepSeek-R1) and leading close-source models (e.g., GPT-5, Gemini-2.5-Pro, Claude-Sonnet-4), achieving state-of-the-art results. In addition, FunReason-MT demonstrates balanced performance across all sub-metrics, indicating strong generalization and stability across reasoning dimensions.

Overall, these results demonstrate that FunReason-MT effectively enhances multi-turn reasoning and tool-use

capabilities by utilizing high-quality synthesized data, resulting in consistent and reliable improvements for real-world function-calling tasks.

5.3 Agentic Results

To investigate the transference of agentic capabilities inherent in the FunReason-MT, we evaluate our models on the OOD BFCLv4 benchmark. Specifically, we assess models on Search and Memory subsets, showing the results in Table 3.

Models	Overall	Web Search			Memory			
		Overall	Base	No Snippet	Overall	KV	Vector	Recursive Sum
ToolACE-2-8B	14.83	9.00	13.00	5.00	20.65	7.74	18.06	36.13
BitAgent-8B	8.24	4.00	4.00	4.00	12.47	4.52	20.65	12.26
watt-tool-8B	6.30	4.00	7.00	1.00	8.60	3.87	13.55	8.39
xLAM-2-3b-fc-r	7.42	1.50	1.00	2.00	13.33	4.52	10.97	24.52
Qwen3-4B-Instruct-2507	8.85	5.00	4.00	6.00	12.69	11.61	11.61	14.84
+FunReason-MT (SFT)	9.99 _{+1.14}	9.00 _{+4.00}	12.00	6.00	10.97 _{-1.72}	1.29	0.00	31.61
+FunReason-MT (RL)	15.10 _{+6.25}	16.00 _{+11.00}	20.00	12.00	14.19 _{+1.50}	5.81	0.00	36.77

Table 3: Performance of on BFCLv4 (last updated August 26, 2025).

On the BFCLv4 benchmark, the FunReason-MT model demonstrates significant performance gains over its base model, Qwen3-4B-Instruct-2507. SFT increases the Overall performance score from 8.85 to 9.99 (+1.14), while RL further boosts it to 15.10 (+6.25). The RL model’s strength is particularly evident in the Web Search Overall subset, where its score raises from 5.00 to 16.00 (+11.00), alongside an increase in the Memory Overall score from 12.69 to 14.19. Notably, the RL-trained FunReason-MT outperforms all other competing specialized models in the main Overall metric.

The FunReason-MT model, particularly after being trained with reinforcement learning, demonstrates significant performance improvements on the BFCLv4 benchmark. This proves that data generated by this paradigm consistently enhances the model’s agentic capabilities. We believe this paradigm of procedural data generation process can lay a strong foundation and provide a robust initial knowledge base for future large-scale, environment-feedback based, agentic reinforcement learning.

6 Conclusion

In this technical report, we introduce FunReason-MT, a novel data synthesis framework designed to address critical bottlenecks in multi-turn function calling data generation. We identified that existing methods, such as random sampling and MAS role-playing, are insufficient, producing data that lacks diversity and fails to capture complex, logically-dependent trajectories. Our framework overcomes these limitations through three core components: Environment-API Graph Interactions for targeted scenario generation, Advanced Tool-Query Synthesis for creating logically complex queries, and a Guided Iterative Chain for robust CoT generation. Empirical validation on the BFCLv3 benchmark demonstrates that our approach achieves state-of-the-art performance among comparable-sized models. Furthermore, our model shows promising out-of-distribution generalization on BFCLv4, underscoring the robustness of our method. This research paves the way for developing more robust and stable models capable of complex reasoning and environment exploration.

References

- Shanshan Han, Qifan Zhang, Yuhang Yao, Weizhao Jin, and Zhaozhuo Xu. Llm multi-agent systems: Challenges and open problems. *arXiv preprint arXiv:2402.03578*, 2024.
- Bingguang Hao, Maolin Wang, Zengzhuang Xu, Yicheng Chen, Cunyin Peng, Jinjie Gu, and Chenyi Zhuang. Exploring superior function calls via reinforcement learning. *arXiv preprint arXiv:2508.05118*, 2025a.
- Bingguang Hao, Maolin Wang, Zengzhuang Xu, Cunyin Peng, Yicheng Chen, Xiangyu Zhao, Jinjie Gu, and Chenyi Zhuang. Funreason: Enhancing large language models' function calling via self-refinement multiscale loss and automated data refinement. *arXiv preprint arXiv:2505.20192*, 2025b.
- Chengrui Huang, Shen Gao, Zhengliang Shi, Dongsheng Wang, and Shuo Shang. Ttpa: Token-level tool-use preference alignment training framework with fine-grained evaluation. *arXiv preprint arXiv:2505.20016*, 2025.
- Zixuan Ke, Fangkai Jiao, Yifei Ming, Xuan-Phi Nguyen, Austin Xu, Do Xuan Long, Minzhi Li, Chengwei Qin, Peifeng Wang, Silvio Savarese, et al. A survey of frontiers in llm reasoning: Inference scaling, learning to reason, and agentic systems. *arXiv preprint arXiv:2504.09037*, 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.
- Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, 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*.
- Akshara Prabhakar, Zuxin Liu, Ming Zhu, Jianguo Zhang, Tulika Awalgona, Shiyu Wang, Zhiwei Liu, Haolin Chen, Thai Hoang, Juan Carlos Niebles, et al. Apigen-mt: Agentic pipeline for multi-turn data generation via simulated agent-human interplay. *arXiv preprint arXiv:2504.03601*, 2025.
- Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv: 2409.19256*, 2024.
- MAOLIN WANG, YINGYI ZHANG, CUNYIN PENG, YICHENG CHEN, WEI ZHOU, JINJIE GU, CHENYI ZHUANG, RUOCHENG GUO, BOWEN YU, WANYU WANG, et al. Function calling in large language models: Industrial practices, challenges, and future directions. 2025.
- An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.
- Xingshan Zeng, Weiwen Liu, Lingzhi Wang, Liangyou Li, Fei Mi, Yasheng Wang, Lifeng Shang, Xin Jiang, and Qun Liu. Toolace-mt: Non-autoregressive generation for agentic multi-turn interaction. *arXiv preprint arXiv:2508.12685*, 2025.
- Guibin Zhang, Hejia Geng, Xiaohang Yu, Zhenfei Yin, Zaibin Zhang, Zelin Tan, Heng Zhou, Zhongzhi Li, Xiangyuan Xue, Yijiang Li, et al. The landscape of agentic reinforcement learning for llms: A survey. *arXiv preprint arXiv:2509.02547*, 2025.
- Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyuan Luo, Zhangchi Feng, and Yongqiang Ma. Llamafactory: Unified efficient fine-tuning of 100+ language models. *arXiv preprint arXiv:2403.13372*, 2024.

A Supplementary Case

This Section details the case study for FunReason-MT. We describe the synthesis process and a complete trajectory.

Case: We showcase the construction of Hard Query with logical jump that directly enables ticket purchase, which involves a jump for consulting the city zip code.

Trace:

```
get_zipcode(city="Rivermist")
output: "83214"
get_zipcode(city="Stonebrook")
output: "74532"
buy_tickets(zipcodeA="83214", zipcodeB="74532")
output: "ticket_id: 14589"
```

Advanced Tool:

buy_tickets_adv(input_param: cityA, cityB, output: ticket_id, desc: purchase air tickets between two cities by city names, returning the purchased ticket information.)

Hard Query:

Please purchase air tickets between the city "Rivermist" and the city "Stonebrook".

Hint:

"purchase air tickets between two cities by city names, returning the purchased ticket information" is equal to "retrieve the zipcode for a given city" and "buy tickets from cityA to cityB with zipcode."

Assistant:

<think>

Okay, so the user wants me to purchase air tickets between Rivermist and Stonebrook. Let me see how I can approach this using the tools provided. First, I need to understand the tools available. There's 'get_zipcode' which retrieves a zipcode for a given city, and 'buy_tickets' which buys tickets using the zipcodes of the start and target cities. Hmm, the user's query is about purchasing tickets between two cities, but the tools require zipcodes. So, I can't directly buy tickets with city names; I need to get their zipcodes first. Alright, step one: use 'get_zipcode' for Rivermist. I'll call it with the city parameter as "Rivermist". This should give me the zipcode for Rivermist. Next, I'll do the same for Stonebrook. Another call to 'get_zipcode' with "Stonebrook" as the city parameter. That should give me the zipcode for Stonebrook. Once I have both zipcodes, I can use the 'buy_tickets' tool. I'll input the zipcodes of Rivermist and Stonebrook into 'buy_tickets' to purchase the tickets. Wait, the output format the user provided is a list of function calls in a specific format. So I need to structure my response with the function names and their parameters correctly. I should make sure that each function call is properly formatted, especially the parameters. Since the city names are strings, I'll enclose them in quotes as per the user's instructions. Let me double-check the parameters for each function. For 'get_zipcode', it's just the city name. For 'buy_tickets', it's both cityA_zipcode and cityB_zipcode. I need to ensure that I pass these correctly in the order specified. I think I've got the sequence right. First, get the zipcodes for both cities, then use those zipcodes to buy the tickets. That way, the tools are used in the correct order to achieve the user's goal. I should also consider if there are any edge cases, like if a city doesn't have a zipcode. But since the user provided specific cities, I'll assume the tools can handle them. Alright, putting it all together, I'll list the function calls in the required format, making sure each parameter is correctly specified with quotes around the string values.

</think>

<tool_call>

```
[get_zipcode(city="Rivermist"), get_zipcode(city="Stonebrook"), buy_tickets(cityA_zipcode="12345", cityB_zipcode="67890")]
```

</tool_call>

Turn:

Query:

Please purchase air tickets between the city "Rivermist" and the city "Stonebrook".

Tools:

get_zipcode, buy_ticket, ...

Assistant:

<think>

Okay, so the user wants me to purchase air tickets between Rivermist and Stonebrook. Let me see how I can approach this using the tools provided. First, I need to understand the tools available. There's 'get_zipcode' which retrieves a zipcode for a given city, and 'buy_tickets' which buys tickets using the zipcodes of the start and target cities. Hmm, the user's query is about purchasing tickets between two cities, but the tools require zipcodes. So, I can't directly buy tickets with city names; I need to get their zipcodes first. Alright, step one: use 'get_zipcode' for Rivermist. I'll call it with the city parameter as "Rivermist". This should give me the zipcode for Rivermist. Next, I'll do the same for Stonebrook. Another call to 'get_zipcode' with "Stonebrook" as the city parameter. That should give me the zipcode for Stonebrook. Once I have both zipcodes, I can use the 'buy_tickets' tool. I'll input the zipcodes of Rivermist and Stonebrook into 'buy_tickets' to purchase the tickets. Wait, the output format the user provided is a list of function calls in a specific format. So I need to structure my response with the function names and their parameters correctly. I should make sure that each function call is properly formatted, especially the parameters. Since the city names are strings, I'll enclose them in quotes as per the user's instructions. Let me double-check the parameters for each function. For 'get_zipcode', it's just the city name. For 'buy_tickets', it's both cityA_zipcode and cityB_zipcode. I need to ensure that I pass these correctly in the order specified. I think I've got the sequence right. First, get the zipcodes for both cities, then use those zipcodes to buy the tickets. That way, the tools are used in the correct order to achieve the user's goal. I should also consider if there are any edge cases, like if a city doesn't have a zipcode. But since the user provided specific cities, I'll assume the tools can handle them. Alright, putting it all together, I'll list the function calls in the required format, making sure each parameter is correctly specified with quotes around the string values.

</think>

<tool_call>

[get_zipcode(city="Rivermist"), get_zipcode(city="Stonebrook"), buy_tickets(cityA_zipcode="12345", cityB_zipcode="67890")]

</tool_call>

Trajectory: We showcase a two-turn complete trajectory.

User:

Determine the current working directory path, and then search for all files and directories in the current directory and its subdirectories.

Assistant:

<think>

Okay, let's tackle the user's request step by step. The user wants to determine the current working directory path and then search for all files and directories in the current directory and its subdirectories.

First, I need to figure out the current working directory. The 'pwd' command is perfect for that. It doesn't require any parameters, so I'll call it first. That should give the user the path they're in.

Next, the user wants to search all files and directories in the current directory and subdirectories. The 'find' function can do this. The 'find' tool has parameters 'path' and 'name'. The default for 'path' is ".", which is the current directory. Since they want everything, the 'name' parameter should be None (or omitted) to return all items. The default for 'name' is already None, so I just need to call find with the default parameters.

Wait, the user specified "its subdirectories" so the recursive search is covered by the find tool's description. So using find with path "." and no name should work. I don't need to specify any parameters except maybe path, but since it's default, it's okay.

So the function calls should be first pwd, then find. Let me check the parameters again. The find function's 'path' defaults to ".", which is correct. The 'name' is optional and defaults to None. So the correct calls are [pwd(), find(path=".)].

I need to make sure that the parameters are correctly formatted. Since 'path' is set to ".", even though it's the default, including it explicitly might be better to ensure clarity. The user might expect the search to start at the current directory, so specifying path="." is safe.

No other functions are needed here. The 'ls' command only lists the current directory, not subdirectories, so 'find' is the right choice. Alright, that should cover the user's needs.

</think>

<tool_call>

[pwd(), find(path=".")]

</tool_call>

Tool:

<tool_response>

```
[('pwd'): {"current_working_directory": "/workspace", 'find(path=".")': {"matches": ["./processed", "./processed/vqrqs3933.md", "./processed/akqff.md", "./processed/documents", "./requirements.txt", "./dqyiu-68828.json", "./ulyuo.json", "./data", "./data/raw", "./data/raw/dtpsp.json", "./data/metadata.json", "./data/vwcc1.md", "./data/unggv.json", "./data/code", "./data/project"]}]
```

</tool_response>

Assistant:

<think>

Okay, let's see what the user needs here. The user asked to determine the current working directory and then search all files and directories in the current directory and subdirectories.

First, the assistant already called pwd() and find(path=".") in the previous turn. The tool responses show that pwd() returned "/workspace" as the current directory. The find function with path "." gave a list of matches including various files and directories under the current directory and its subdirectories.

Now, the user's requirement is to address their query in the next step. Since the user wanted both the current directory and a search, the assistant already executed both functions successfully. The tool_response includes the path and the matches.

The thought process should confirm that both functions were executed and their results are available. The reply should describe the results. Since the user's needs were met, there's no need for further function calls. The assistant should present the findings, like stating the current directory and listing the found items. But according to the output format, the reply_to_the_user should either be function calls or a description. Since the functions are already done, the reply should be a description summarizing the results.

Wait, the output format says if you need to give a description, output a sentence. So the assistant should say something like "The current working directory is /workspace. The search found the following items: [list of matches]." But the matches are in the tool_response's find result. However, the user might just need the info presented.

But looking back at the instructions, in the reply_to_the_user section, if you need to give a description, output a sentence. Since the functions have been executed, the next step is to describe the results. So the assistant should format the answer with the current directory and the matches from find.

Wait, but the example given in the instructions for reply_to_the_user when giving a description is like "the cd function is executed success, and current workdirctory is src." So in this case, the assistant should combine both results.

So the thought process would note that both functions were executed successfully. The reply should state the current directory and list the found items. But since the matches are many, maybe just mention that the search found items

like those listed.

Alternatively, the exact answer should reference the tool_response data. The current_working_directory is /workspace, and the matches array includes all those paths. So the assistant can present that info concisely.

</think>

<tool_call>

The current working directory is /workspace. The search found the following items: ["./processed", "./processed/vqrqs3933.md", "./processed/akqff.md", "./processed/documents", "./requirements.txt", "./dqyiu_68828.json", "./ulyuo.json", "./data", "./data/raw", "./data/raw/dtsp.json", "./data/metadata.json", "./data/vwcc1.md", "./data/unggv.json", "./data/code", "./data/project"]."

</tool_call>

User:

Display the contents of the file named unggv.json in the data directory, and then display the last 7 lines of the file named dtsp.json, which is in the data directory of the raw directory.

Assistant:

<think>

Okay, let's tackle the user's latest request. They want to display the contents of "unggv.json" in the data directory and the last 7 lines of "dtsp.json" located in the data/raw directory.

First, I need to check the current directory. From the previous tool response, the current working directory is "/workspace". The user mentioned "data directory", so "unggv.json" is directly under "/workspace-/data/unggv.json". But to access it, I should use the 'cat' command. However, 'cat' requires the file to be in the current directory. Since the current directory is "/workspace", I need to check if "unggv.json" is there. Wait, looking back at the find results, "unggv.json" is listed under "./data/unggv.json". So it's in the data subdirectory. To access it, I need to either change into the data directory first or specify the path, but the 'cat' tool only allows the file name in the current directory. Therefore, I should first 'cd' into the data directory.

Wait, the 'cd' tool changes the directory, so first step: cd into 'data'. Then, use 'cat unggv.json'. After that, for the second part, the user wants the last 7 lines of 'dtsp.json' in data/raw. Since we are now in the data directory, the 'raw' directory is a subdirectory. So we need to 'cd raw' to get into that directory, then use 'tail' with lines=7 on 'dtsp.json'. But after that, we might need to go back, but the user didn't ask for that. Alternatively, maybe we can do it without changing directories again by specifying the path, but according to the tool's parameters, 'tail' only takes the file name in the current directory.

Wait, the 'tail' parameters say "No path is allowed and you can only perform on file at local directory." So to access 'data/raw/dtsp.json', we need to be in the 'data/raw' directory. So the steps would be:

1. Current directory is /workspace. First, to access 'unggv.json' in data/, do 'cd data', then 'cat unggv.json'. 2. Then, to get into data/raw, do 'cd raw', then 'tail(file_name="dtsp.json", lines=7)'.

But after the first 'cd data', the current directory becomes /workspace/data. Then after 'cd raw', it becomes /workspace/data/raw. Then after those commands, the user might want to return, but since it's a multi-step process, the assistant should just proceed step by step.

However, the user's request is to first display unggv.json and then dtsp.json. So the function calls would be:

First, cd into data: [cd(folder="data")]

Then, cat unggv.json: [cat(file_name="unggv.json")]

Then, cd into raw: [cd(folder="raw")]

Then, tail dtsp.json with 7 lines: [tail(file_name="dtsp.json", lines=7)]

But after these steps, the current directory is now data/raw. The user might need to go back, but since the user didn't ask, maybe it's okay. Alternatively, maybe there's another way without changing directories multiple times, but according to the tool constraints, it's necessary.

Wait, but the 'find' result shows that 'dtsp.json' is in "./data/raw/dtsp.json". So the path from the current directory (which is /workspace) would require navigating into data/raw. So the steps are necessary.

Therefore, the function calls should be in the order:

1. cd into data 2. cat unggv.json 3. cd into raw 4. tail dtsp.json with lines=7

So the function calls would be:

[cd(folder="data"), cat(file_name="unggv.json"), cd(folder="raw"), tail(file_name="dtsp.json", lines=7)]

But I need to make sure each step is correct. Let me check each function's parameters again.

For 'cd', the parameter is 'folder', which is the directory to change to. So first cd to "data", then from there cd to "raw" (since current is now data/).

The 'cat' requires the file name in current directory. After cd data, unggv.json is there. Then, after cd raw, dtsp.json is in current directory. So yes, that works.

Therefore, the correct function calls are as above.

</think>

<tool_call>

[cd(folder="data"), cat(file_name="unggv.json"), cd(folder="raw"), tail(file_name="dtsp.json", lines=7)]

</tool_call>

Tool:

<tool_response>

```
[`cd(folder="data")': "current_working_directory": "data", 'cat(file_name="unggv.json)": "file_content": " "data": "id": "94", "type": "product" ", 'cd(folder="raw")': "current_working_directory": "raw", 'tail(file_name="dtpsp.json", lines=7)': "last_lines": " "data": "id": "27", "type": "user" "}]
```

</tool_response>

Assistant:

<think>

Okay, let's see what the user is asking for now. The user's latest query was to display the contents of "unggv.json" in the data directory and the last 7 lines of "dtpsp.json" located in the data/raw directory.

Looking at the previous interactions, the assistant first used pwd() and find() to get the current directory and list all files. Then, the user's next request required navigating into the data directory with cd("data"), then using cat on unggv.json. After that, they changed to the raw directory with cd("raw") and used tail on dtpsp.json with lines=7. The tool responses show that all these commands were executed successfully. The current working directory was first changed to "data", then to "raw". The cat and tail commands both returned the expected file contents. Since all the functions called in the assistant's response have been executed without errors, there's no need to repeat any commands. The user's requirements have been fully met. Now, the assistant should present the results of these commands to the user in a clear manner, mentioning the contents of unggv.json and the last lines of dtpsp.json as per the tool responses.

</think>

<answer>

The contents of "unggv.json" in the data directory are: " "data": "id": "94", "type": "product" ".

The last 7 lines of "dtpsp.json" in the data/raw directory are: " "data": "id": "27", "type": "user" ".

</answer>