MAGNET: Multi-turn Tool-use Data Synthesis and Distillation via Graph Translation

Anonymous ACL submission

Abstract

We propose MAGNET, a principled approach to synthesizing high-quality training trajectories to enhance the function calling capability of large language model agents in multi-turn conversations with humans. To reflect the complicated function interactions in multi-turn cases, we take a graph perspective and design novel node operations to build reliable signature path of functions. We iterative transform the signature path into pairs of multi-turn user queries and executable FCs. Motivated by context distillation, we propose to leverage the pairs of query-FC to sample positive trajectories from a teacher model with the references as context and negative trajectories that contrast with the positive ones on targeted error types. Experiments show that our 14B model trained with the positive trajectories with supervised fine-tuning and preference optimization against negative trajectories, MAGNET-14B-mDPO, obtains 68.01 on BFCL-v3 and 73.30 on ToolQuery, surpassing the performance of the teacher model Gemini-1.5-pro-002 by a large margin.

1 Introduction

Autonomous agents based on large language models (LLMs) have made remarkable progress on fulfilling complex agentic tasks (Yin et al., 2024; Ma et al., 2024; Zhang et al., 2024), benefiting from the high capacity of reasoning and planning of LLMs (Achiam et al., 2023; Team et al., 2024; Hui et al., 2024). Among the skillset for agents, the ability to leverage external tools or application programming interfaces (APIs) [1] and interact with humans to perform actions in environments is in the central of successful completion of many agentic tasks. Towards this end, recent LLMs have been tailored for function calling (FC) abilities (Schick et al., 2023; Patil et al., 2023; Dubey et al., 2024; Yang et al., 2024), achieving improved performance on benchmarks that simulate real-world APIs (Yan et al., 2024; Yao et al., 2024; Guo et al., 2024; Ma et al., 2024).

However, by qualitatively scrutinizing the behaviors of models, we find that despite the advancements in composing independent FCs, it is still challenging for current LLM agents to perform multi-step and multi-turn interactions with users [2] where LLM agents reason, compose FCs and analyze outputs from FCs to respond (Yao et al., 2024; Yan et al., 2024). We summarize three main challenges and common mistakes in multi-turn FC, as illustrated in Figure 1: 1) *Nested FCs*: some turns require multiple or even nested FCs which might not be explicitly requested in the query; 2) *Long dependency*: some turns require information from the conversation history to compose FCs; 3) *Irrelevance*: some turns might contain missing functionality or parameter values, for which additional clarification questions are required. Performance-wise, in the Berkeley Function Calling Leaderboard (BFCL-v3) (Yan et al., 2024), the best proprietary model achieves 47.62% success rate on multi-turn cases, while some public models have only around 10% success rate.

Synthesizing or distilling data from stronger LLMs has been proven a powerful way to improve the reasoning abilities (Guo et al., 2025) of weaker LLMs. Yet the limited performance of existing models on multi-turn cases aggravates the difficulty in gathering high-quality training trajectories to improve the multi-turn ability of public models. To bridge the gap between single FC and multi-turn interactions and build reliable trajectories, we propose a principled pipeline, called Multi-turn function-cAlling data synthesis with Graph

[1] The terms, *function calling* and *tool-use*, *function* and *API*, are used interchangeably in this paper.

[2] Multi-step interactions require the LLMs to execute multiple internal FCs to address a single user request, while multi-turn interactions involve an extended exchange between the user and the agents, resulting in multiple conversational turns.

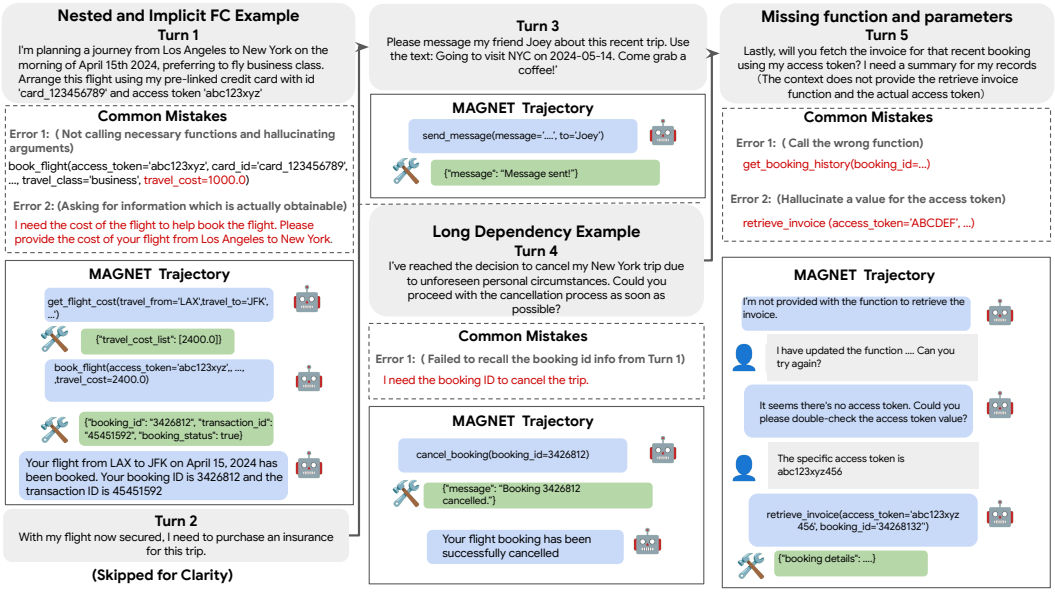

Figure 1: Illustration of challenges and common mistakes in multi-turn FC. An agent needs to understand function outputs and finish follow-up queries from users. This brings several challenges to the agent such as nested FCs (turn 1), long output dependencies (turn 4), irrelevant functions (turn 5).

Translation, or MAGNET, to collect trajectories to train public models with both supervised fine-tuning (SFT) and preference optimization.

Our method is based on iterative *back-and-forth translation* (Section 3.3). Given a sequence of function signatures, i.e., function names and documentations, we prompt LLMs to iteratively translate them into queries, mimicking user requests, and then compose executable FCs as references. However, forming the *function signature path* (FSP) is not straightforward. Previous works (Qin et al.) focus on single-turn FCs and randomly sample functions from the same domains. We propose a graph-based perspective to construct multi-turn FSPs.

Motivated by the fact that two functions from the same domain are likely to be relevant in terms of their inputs and outputs. Therefore, we organize functions as nodes in a graph structure and set a directed edge between two nodes when the source node's outputs relate the target node's inputs. We call them *local dependency graph* as the edges reflect the dependencies among functions. Based on the local dependency graph, we random walk to sample related function signatures and form a FSP.

In the graph-level, we find that those challenges mentioned in Figure 1 can be abstracted as node operations. For example, nested FCs can be abstracted as `Insert`, which adds extra nodes before another node. Therefore, we further design three node operations: `Insert`, `Merge`, `Split` to enhance the initial FSPs and tailor them to cover the challenges. We show through qualitative (Figure 1) and ablation study (Section 4) that including those operations largely improve the reasoning process and reduce common mistakes in multi-turn challenges.

Given the queries and FC references pairs as additional signals, we further control the trajectory generation process with Gemini-1.5-pro-002 as the teacher LLM using context distillation (Snell et al., 2022). Specifically, we add FC references as hints while synthesizing trajectories to ensure the quality of positive trajectories. To enable preference-based optimization, we also construct negative trajectories by selecting actions that are making mistakes and deliberately add wrong hints with those mistakes into the trajectories. This makes clear contrary between positive and negative trajectories.

Experiments on two common benchmarks, BFCL-v3 and ToolQuery, demonstrate the advantage of our pipeline. By SFT Qwen2.5-Coder models on 34K trajectories and on 4,556 trajectory pairs with multi-turn direct preference optimization (mDPO) (Xiong et al., 2024), our model MAGNET-14B-mDPO achieves rank 4th on the BFCL-v3 benchmark, surpassing proprietary models like o1, GPT-4-turbo, and the teacher model, Gemini-1.5-pro-002, adding to the base model by 32.5 points on multi-turn cases. Ablation study further shows that all the components in the pipeline is helpful to improving its capability, and the performance im-

provement can generalize to different base models.

Our contributions can be summarized as follows:

- A graph-based perspective for constructing and enhancing high-quality multi-turn queries and FC references, covering the challenges in multi-turn FCs.
- A novel technique to distill the information provided in FC references to construct training trajectories for both SFT and mDPO.
- We demonstrate superior performance on BFCL-v3 and ToolQuery benchmarks with public models trained with our data. Detailed ablation study shows the effectiveness of each component.

2 Related Work

FC agents evaluation The ability to use external tools to solve a complex task when the agent lacks some knowledge intrinsically is crucial in agentic behaviors. A variety of benchmarks have been constructed to evaluate such ability. We roughly categorize them as follows based on the amount of functions needed for each test instance and the interactions among functions: (1) single-step; (2) multi-steps, which can be further decomposed into parallel, multiple (chained but not nested), nested; (3) multi-turns. Among those, BFCL-v3 (Yan et al., 2024) is a comprehensive benchmark evaluating single-step, multi-steps, multi-turns scenarios. NexanRaven [3], Toolbench (Qin et al.), StableToolbench (Guo et al., 2024) mainly test for multi-steps tool-use. Basu et al. (2024) target nested API calls. Yao et al. (2024); Ma et al. (2024); Lu et al. (2024) feature multi-turns and multi-steps FCs. However, most of the above mentioned datasets are human curated (with the assistant of LLMs).

Training FC agents Due to the lack of training trajectories, fine-tuning a tool-use agent typically starts with collecting training data. Toolformer (Schick et al., 2023) replaces segments in texts with API calls to train LLMs. (Qin et al.; Chen et al., 2024b) synthesize queries from random sampled APIs without clear structure. The xLAM and APIGen series (Liu et al., 2024b; Zhang et al., 2024) unify the format of tool-use data with other agentic tasks and automatically generate queries from verified APIs. Lin et al. (2024) improves the APIGen (Liu et al., 2024b) dataset and propose to add function masking and more irrelevant functions to improve the robustness of agents. Abdelaziz et al. (2024) introduce fine-tuning with multi-task (function calling, instruction tuning) on 110k data. Liu et al. (2024a) synthesize new APIs automatically and directly prompts LLMs to role-play users, agents, and tools. Chen et al. (2024a) adapt composition to improve the quality of single-turn function calling. Among those works, Qin et al.; Chen et al. (2024b); Liu et al. (2024b) back-translate queries from APIs. While our query generation technique adopt similar ideas, to adapt to multi-turn cases, we propose to organize function signatures in graphs and apply node operations to improve graph complexity. Our trajectory synthesis methods also diverge from previous methods by incorporating more controls.

[3] https://nexusflow.ai/blogs/ravenv2

3 Methodology: MAGNET

In this section, we first discuss the whole training pipeline to provide more context (Section 3.1). Then, we dive into our main contribution of synthesizing high-quality FC trajectories (Section 3.2).

3.1 Training setup and formulation

We leverage the typical two-stage SFT + RLHF training. In the first stage, suppose we have a base model and a set of training trajectories $\{\tau^i\}$, $i = 1 \ldots n$. Each trajectory involves a sequence (k-steps) of user queries, model actions, and tool responses: $\tau_w^i = \left(q_1^i, a_1^i, t_1^i \ \cdots \ q_k^i, a_k^i, t_k^i\right)$. The SFT training uses maximum likelihood estimation (MLE) to fit the model actions $a_1^i \cdots a_k^i$, i.e., the blue parts in Figure 1, given the rest as context. Then, in the second stage, given the SFT model and trajectory pairs $\{\tau_w^i, \tau_l^i\}$, $i = 1 \ldots m$, we adopt the mDPO loss (Xiong et al., 2024) and a MLE loss to further tune the SFT model:

$$\mathcal{L}\left(x; \tau_w, \tau_l\right) = \mathcal{L}_{\text{SFT}}\left(x; \tau_w\right) + \lambda \mathcal{L}_{\text{mDPO}}\left(x; \tau_w, \tau_l\right),$$
$$\mathcal{L}_{\text{mDPO}}\left(x; \tau_w, \tau_l\right) =$$
$$-\log\sigma\left(\eta\left(\sum_{\tau_l} \frac{\pi_\theta\left(a^l|s^l\right)}{\pi_{ref}\left(a^l|s^l\right)} - \sum_{\tau_w} \frac{\pi_\theta\left(a^w|s^w\right)}{\pi_{ref}\left(a^w|s^w\right)}\right)\right), \quad (220)$$

where τ_l and τ_w represents the negative and positive trajectories. λ is a weight hyperparameter for balancing the two losses, π_{ref} is the SFT reference policy, and π_θ is the mDPO policy. Next, we dive into MAGNET, our method that synthesizes $\{\tau^i\}, i = 1 \ldots n$ and $\{\tau_w^i, \tau_l^i\}, i = 1 \ldots m$.

3.2 Data Construction Overview

Overall, MAGNET first generates pairs of queries and FC references, and then, transform them into

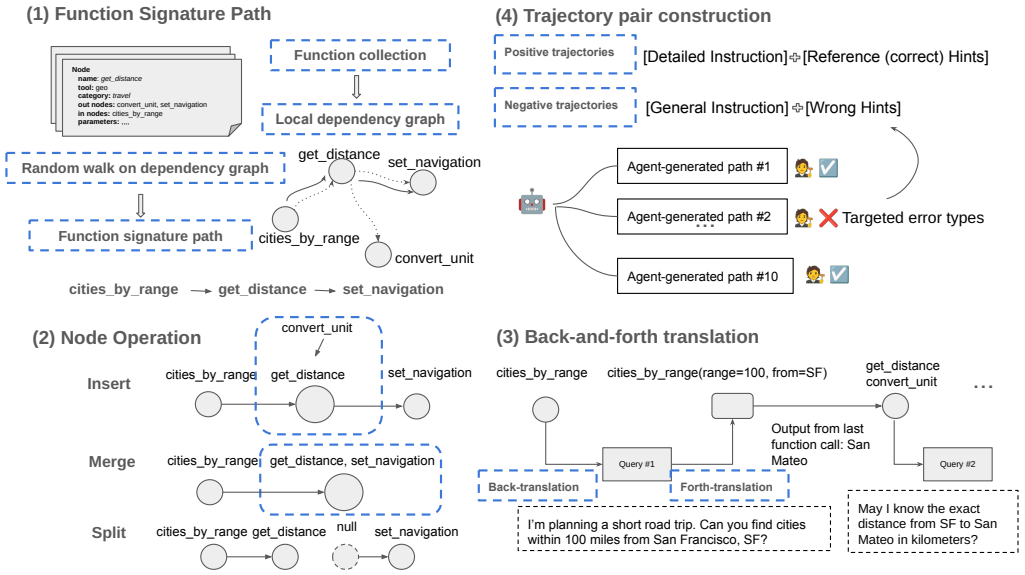

Figure 2: The pipeline for constructing trajectories of function calling. We divide the pipeline into four parts and depicts each part respectively. (1) Construction of the function pool and function execution graph; (2) Node operations defined on the function execution graph; (3) Back-and-forth translation to iteratively create multi-turn queries and fill in function parameters; (4) Construction of positive and negative trajectories by context distillation of good and bad hints and instructions.

trajectories using a teacher LLM. In the first part, the backbone is a back-and-forth translation process inspired by Nguyen et al. (2024); Li et al. (2024) that converts FSPs into query-reference pairs, which will be introduced in section 3.3. The key innovation, however, lies in how we construct high-quality multi-turn FSPs with a graph perspective and node operations (Section 3.4 and 3.5). In the second part, we collect both positive and negative trajectories with our newly designed *context distillation* technique. An illustration in the actual order of the pipeline is in Figure 2. For the whole process, we prompt an LLM to help us on tasks like rewriting, back-and-forth translation etc. Without extra statement, we will use Gemini-1.5-pro-002 as the assistant LLM. All the prompts mentioned in this section are in Appendix A.

3.3 Back-and-forth translation

We start with the back-and-forth translation process. Suppose we have obtained a FSP, noted as $\phi = (f_1, f_2, \cdots f_k)$, where each $f_i, i = 1, 2, \cdots, k$ denotes function signatures used in turn i. In this stage, we create the pair of user queries and executable FC references (q_i, a_i) with each f_i. For back-translation, given the function signatures of turn i, f_i, we prompt the assistant LLM to generate q_i that involves real information for the parameters. Then, for forth-translation, we prompt the assistant LLM to convert information in the query into parameter values for the given functions to produce a_i. This process is conducted iteratively for each function signature in the FSP to make sure that the outputs from the previous rounds are ready before passing to the later rounds, as illustrated in (3) in Figure 2. From the next section, we highlight how to generate the FSPs, ϕ.

3.4 Function collection and initial FSPs

Our function collection inherits from previous works (Liu et al., 2024b). We collect the underlined function source codes from the StableToolBench (Guo et al., 2024) and BFCL-v3 multi-turn function implementation (Yan et al., 2024). For functions in BFCL-v3, we rewrite the function name and descriptions using our assistant LLM, i.e., only the real implementation that is not exposed to models are kept. For StableToolBench, following APIGen, we select those that contain parameters and are executable verified by simulated calls. In total, we collect 5,011 APIs. Each function in the pool will be viewed as a node. For node attributes, we prompt the assistant LLM to label their category and class. For example, a function like `get_current_weather` will have category `Weather` and the tool class `Weather condition tool`. Our categories come from StableToolBench, which include 49 categories.

Then, we construct local dependency graph on functions from the same category and class. For each target node, we sample 30 candidate nodes from the same tool class and same category and prompt the assistant LLM to judge whether there are dependencies between the target node and candidate nodes based on their inputs and outputs. If relevant, we will add a directed edge from the target node to that candidate node.

Finally, starting with a random node, we conduct a random walk to sample subsequent function calls. The random walk ends when the step reaches a predefined maximum step, which we set as seven. After the random walk, we will collect an initial FSP, which consists of function signatures, i.e, function names only, shown in part (1) of Figure 2.

3.5 Node operations for enhanced FSPs

To better cover the challenges for multi-turn interactions, we propose to enhance the FSPs obtained above with graph-level operations.

Node OP #1: `Insert` is designed for handling the nested and implicit function call and long dependency scenario. Consider the query:

`Please check how many kilometers to go from San Francisco to San Mateo,`

which should invoke two functions:

`get_distance(from_loc,to_loc)`

`convert_unit(in_value=<milage obtained from SF to SM>, out_value)`.

The first function will return a distance in mileage and we need the second function to convert them into kilometers. However, the second function is not mentioned explicitly in the query to be called. Models might not recognize to call the second function. To cover this, our `Insert` operation will insert an implicit function signature into the current FSP if they are nested. Specifically, for a target node in FSP, we check for premise function signatures in the function pool using our assistant LLM (see the prompts to judge nested functions in Appendix A), then, we insert a random premise function signature into the target node in FSP.

`Insert` will also be useful for creating examples covering the long dependency challenge. For example, we could `Insert` another `cities_by_range(range=)` in a few rounds later which reuses the outputs from `get_distance(from_loc,to_loc)`.

Node OP #2: `Merge` is for creating a single-turn query that would involve multiple function calls and cover short dependency. Notice that the key difference with `Insert` and nested API calls is that we could `Merge` multiple functions that are relevant but not exactly nested. In this case, agents should understand the outputs from the previous functions in this turn to compose the consecutive function. For example, the following query would invoke both `get_distance(from_loc,to_loc)`, `set_navigation(distance)`:

`Can you check how many kilometers to go from San Francisco to San Mateo and then set up the navigation for me with the obtained distance?`

Node OP #3: `Split` is mainly designed for the missing or irrelevant function information scenarios. For the previous query, if the function `get_distance` is not provided, or the query omits the destination: `Please check how many kilometers to go from San Francisco to somewhere`, the agent should ask a clarification question. We will create a `null` node with a 'miss params' or 'miss func' labels which will act as an indicator when translating.

3.6 Positive and negative trajectory sampling

To transform (q_i, a_i), $i = 1, 2, \cdots, k$ generated in back-and-forth translation into trajectories, we adopt a novel technique assembles to context distillation (Snell et al., 2022). When generating positive trajectories, we hope the teacher model to be as accurate as possible. However, none of current LLMs can consistently produce perfect trajectories. So, we propose to add a `[Hint]` section after each turn to indicate the functions being called during this turn, using the reference FCs and provide detailed instructions when sampling the positive trajectories, to produce high-quality trajectories. We show the prompts in Appendix A.

On the contrary, when generating negative trajectories, we hope the trajectories reflect the mistakes made by models. So for negative trajectories, we also include such hints but the actual content is a misleading wrong FCs. Those FCs are collected from the mistakes of the SFT model. Specifically, for each data instance, we collect ten trajectories from the SFT model. Then, for each turn in each trajectory, we present it to the assistant LLM as a judge to decide whether this turn includes an incorrect FC that conforms with any one of the errors defined in the judgement prompt. If so, we will collect them as misleading hints when prompting SFT model to sample negative trajectories.

3.7 Post-processing and data mixture

We adopt the following post-processing techniques to enhance the diversity of the SFT datasets so that models trained with our data could be more robust to variations in superficial features.

- For each training data, we shuffle the order of available functions in system prompts.
- We filter out trajectories with rule-based metrics: we collect several key words that indicate failed FCs at the end of each turn, such as 'Bad request'. 'does not match' etc. This roughly excludes incorrect trajectories or wrong formatting in FCs.
- Besides multi-turn data, we add the following types of data into our final SFT data mix: single-turn data including those that invoke single, parallel (same function with different arguments), and multiple (different but relevant) FCs. This is for warming up the model on function calling; 2) irrelevance functions where models should be able to detect. A study on how to mix those data is provided in Section 4.4.

3.8 Data statistics

Category	# SFT	# mDPO
Single-turn	20,000	1,556
Multi-turn	7,800	2,250
Irrelevance	6,200	750
Avg. FCs (for single-turns)	1.80	1.94
Avg. Turns (for multi-turns)	4.71	5.22
Avg. FCs (for multi-turns)	15.13	14.98

Table 1: Data statistics for the training sets. # SFT and #mDPO represents the number of samples in SFT and mDPO training sets of the corresponding category.

Our final SFT training set contains 34,000 instances and the preference learning set contains 4,556 instances. The total training size, 38,556, is around half of other current public datasets such as APIGen (60,000), Hammer (67,500) etc. We present a detailed statistics about the number of each data type, the number of turns, and the number of function calls in Table 1.

4 Experiments

We conduct experiments on the following two benchmarks: BFCL-v3 (Yan et al., 2024) and ToolQuery (Ma et al., 2024). BFCL-v3 is a comprehensive benchmark designed for different aspects of function calling, including single-turn, multi-step, multi-turn, and irrelevant function calls categories. ToolQuery is part of a broader agent benchmark that test model's ability in composing multi-step and multi-turn function calls in academia, weather, movie areas. BFCL-v3 have in total 4,751 test cases while ToolQuery contains 60 test cases. We use a unified prompt format for both tasks, as shown in Appendix A.

4.1 Setup

We fine-tune Qwen2.5-Coder-7B-instruct and Qwen2.5-Coder-14B-instruct. For the training, we first train with the 34,000 positive trajectories with SFT. We set a peak learning rate of 1e-5 with warm up and linear decay, and a batch size of 64. Then, in the mDPO stage, we do full fine-tuning on the 7B models and set the learning rate to be 5e-7 and batch size 32. For mDPO on 14B model, we conduct LoRA tuning (Hu et al., 2022) with a learning rate of 1e-6. More details in Appendix B.

4.2 Main results on BFCL-v3

We compare the performance of our trained model with top ranked and related models on the BFCL-v3 benchmark. Results are presented on Table 2. The performance of our best 14B model ranks 4th on the leaderboard, surpassing the o1 model and on par with GPT-4-Turbo on both the overall performance and the multi-turn performance. We show that with mDPO on targeted loss patterns, the performance on multi-turn scenarios can be boosted compared to SFT only models, with a margin of 2.50% success rate for the 14B model. Notice that all of our 7B and 14B models, including SFT and mDPO models, outperform the teacher model Gemini-1.5-pro-002 on the multi-turn scenario. This demonstrates that our data synthesis pipeline introduces additional signals and provides better supervision compared to directly distilling from the teacher model.

Finally, comparing with base models and other public models of the same size, our trained model boosts the performance by 18.5 and 30.0 on multi-turn scenarios for the base 7B and 14B Qwen2.5-Coder models, respectively. We also outperforms Hammer2.1-7b (FC), a competitive FC agent model trained from the same base model.

4.3 Main results on ToolQuery

Results for ToolQuery are shown in Table 3. We achieve a success rate of 73.3 on ToolQuery by training Qwen2.5-Coder-14B-instruct on our data, surpassing the performance of a strong proprietary model, GPT-4o, and a much larger public model

Model	Overall	Single Turn			Multi-turn					Hallucination Measure	
		Non-live AST	Non-live Exec	Live AST	Overall	Base	Miss Func	Miss Param	Long	Relevant	Irrelevant
Top six models											
WATT-TOOL-70B (FC)	74.31	84.06	89.39	77.74	58.75	67.50	57.50	48.50	61.50	94.44	76.32
GPT-4O-2024-11-20 (PROMPT)	72.08	88.10	89.38	79.83	47.62	59.00	41.00	35.50	55.00	83.33	83.76
GPT-4O-2024-11-20 (FC)	69.58	87.42	89.20	79.65	41.00	62.50	6.00	37.50	58.00	83.33	83.15
GPT-4-TURBO-2024-04-09	67.88	84.73	85.21	80.50	38.12	54.00	13.50	35.50	49.50	72.22	83.81
WATT-TOOL-8B* (FC)	67.33	86.44	87.73	76.23	38.25	46.00	40.00	27.00	40.00	77.78	82.89
O1-2024-12-17 (PROMPT)	66.73	78.92	82.70	78.14	28.25	40.50	5.00	34.50	33.00	61.11	89.62
Gemini models (teachers)											
Gemini-1.5-Pro-002 (Prompt)	62.19	88.58	91.27	76.72	20.75	23.00	19.50	17.50	23.00	72.22	78.15
Gemini-2.0-Flash-Exp (Prompt)	61.74	89.96	79.89	82.01	17.88	28.00	3.00	19.00	21.50	77.78	86.44
7B models											
Functionary-Small-v3.1 (FC)	56.49	86.75	87.12	73.75	10.12	18.00	2.50	14.00	6.00	77.78	70.89
Hammer2.1-7b (FC)	61.83	88.65	85.48	75.11	23.50	35.50	**25.50**	19.00	14.00	82.35	78.59
Qwen2.5-Coder-7B-Instruct	53.13	86.83	82.27	66.99	8.25	11.50	6.50	5.50	5.50	**88.89**	65.39
MAGNET-7B-SFT	62.73	88.60	85.73	74.19	26.50	35.50	24.00	**27.50**	19.00	66.67	**78.67**
MAGNET-7B-mDPO	**64.64**	**89.40**	**89.27**	**77.92**	**27.75**	**39.00**	24.00	26.00	**22.00**	83.33	78.51
14B models											
Qwen2.5-Coder-14B-Instruct	51.88	**90.94**	87.80	65.30	5.38	7.50	7.00	4.00	3.00	**100.00**	44.58
MAGNET-14B-SFT	66.83	90.02	88.20	77.92	33.38	47.00	32.00	32.00	22.50	72.22	82.59
MAGNET-14B-mDPO	**68.01**	90.13	**89.75**	**79.14**	**37.88**	**52.00**	**36.00**	**35.50**	**28.00**	88.89	**84.78**

Table 2: Main results on BFCL-v3. Our MAGNET series demonstrate substantial improvements compared to their base model, Qwen2.5-Coder series, in both the multi-turn function calling and overall evaluations. Our 14B model ranked #4 in the leaderboard, surpassing o1 and the teacher model Gemini-1.5-pro-002. Best numbers under each test category and base models are **bold**. * indicates reproduced results with the exact same process as our models.

	Success rate	Progress rate
Qwen-Coder-7B-instruct	15.0	34.0
Qwen-Coder-14B-instruct	51.7	68.7
GPT-4o	63.3	**80.1**
Gemini-1.5-pro-002	68.3	74,6
xLAM-8x22b-r	68.3	75.8
MAGNET-7B-mDPO	67.7	73.4
MAGNET-14B-mDPO	**73.3**	78.7

Table 3: Main results on ToolQuery. Our 14B model achieved the best performance on success rate.

tuned on the function-calling task, xLAM-8x22b-r. Notice that all the functions from ToolQuery are unseen in the training set. This further demonstrates the generalization ability of our trained models on unseen functions.

4.4 Ablation Study and Analysis

We conduct ablation study to answer the questions: (1) how each component in our pipeline affects the overall performance? (2) how our synthetic data is better than other public training datasets? (3) is the effects of the synthetic data consistent among different base models? The full results are presented in Table 4. Findings below:

Pipeline design We conduct experiments to see the effects of local dependency graph construction, each node operation, positive trajectories sampled with correct hints, and negative trajectory sampled with wrong hints in the model performance. As shown in the first part (first six rows) in Table 4, we demonstrate that each component is helpful in the final performance of the model. Especially, with the initial local dependency graph, we are able to improve upon the base model by around 8% on multi-turn success rate. Building upon that, both merge and insert operations boost the multi-turn performance by a large margin, especially on the base multi-turn test cases. Finally, adding split operation directly helps with the missing function, missing parameters, and irrelevance detection scenarios 5.5%, 7.5%, and 3.69%, respectively. We also observe a substantial boost in performance when we distill FC references into positive trajectories compared to directly distilling Gemini-15-pro-002 trajectories from the multi-turn queries. This brings a 14.50% gain in multi-turn performance. Finally, adding negative trajectories using our context distillation technique brings around 0.5% improvements compared to randomly sample rejected trajectories from the SFT model.

Data sources To demonstrate the benefits of our constructed data against other public training data, we train the same base model using different sources of open-sourced data. Specifically, amongst the top-performance models, the only open-sourced training datasets are APIGen (Liu et al., 2024b) and a subset of ToolAce (Liu et al., 2024a). Further, the Hammer2.1-7b model (Lin et al., 2024), although not open-sourcing the full training data, is trained from the same base model with an augmented dataset with irrelevant functions and masking techniques. Therefore, we compare our model with two other models: the same based model trained with a combination of all open-sourced training data, i.e., APIGen and ToolAce, and Hammer2.1-7b (FC). As shown in the second

Model	Overall	Single Turn			Multi-turn					Hallucination Measure	
		Non-live AST	Non-live Exec	Live AST	Overall	Base	Miss Func	Miss Param	Long	Relevant	Irrelevant
Ablation on pipeline components											
init graph	58.54	89.60	87.13	76.96	12.75	14.50	13.00	13.50	10.00	94.44	78.95
+ merge	60.83	89.76	87.81	76.92	20.63	26.50	18.00	19.00	19.00	77.78	76.87
+ merge + insert	64.39	90.89	87.91	77.37	29.25	42.00	26.50	24.50	24.00	88.89	78.90
MAGNET-14B-SFT	66.83	90.02	88.20	77.92	33.38	47.00	32.00	32.00	22.50	72.22	82.59
MAGNET-14B-SFT	66.83	90.02	88.20	77.92	33.38	47.00	32.00	32.00	22.50	72.22	82.59
- context-distillation-positive	60.26	88.27	84.29	76.63	18.88	21.00	20.00	15.50	19.00	72.22	78.00
MAGNET-14B-mDPO	68.01	90.13	89.75	79.14	37.88	52.00	36.00	35.50	28.00	88.89	84.78
- context-distillation-negative	67.35	90.34	88.96	78.84	36.25	48.50	34.50	35.00	27.00	88.89	83.79
Comparison between training data source: Qwen-Coder-7B-instruct as base model											
MAGNET-7B-SFT	62.73	88.60	**85.73**	74.19	26.50	**35.50**	24.00	27.50	19.00	66.67	78.67
APIGen + ToolAce	50.30	88.85	89.59	59.04	7.13	10.50	6.50	5.50	4.50	100.00	39.17
APIGen + ToolAce + Irrelevant	57.24	87.44	89.54	76.99	6.25	9.00	5.50	7.00	3.50	77.78	83.79
Hammer2.1-7b (FC)	61.83	88.65	85.48	75.11	23.50	35.50	25.50	19.00	14.00	b82.35	78.59
Effectiveness of MAGNET across different base models											
QWEN2.5-CODER-INSTRUCT	50.01	86.15	82.45	64.46	4.25	6.00	6.50	3.50	1.00	100.00	51.60
MAGNET-QWEN2.5-CODER-INSTRUCT	62.73	88.60	85.73	74.19	26.50	**35.50**	24.00	27.50	19.00	66.67	78.67
QWEN2.5-INSTRUCT	52.58	86.83	82.27	66.99	7.25	8.50	10.00	5.50	5.00	88.89	65.39
MAGNET-QWEN2.5-INSTRUCT	59.84	88.12	85.48	72.86	21.12	31.00	19.00	21.00	13.50	83.33	76.67
MIXTRAL-8X7B-INSTRUCT-V0.1	36.93	47.94	51.59	57.71	0.50	1.00	0.00	0.00	1.00	38.89	75.37
MAGNET-MIXTRAL	58.17	88.46	80.20	68.46	19.75	24.50	22.50	19.00	13.00	94.44	66.47

Table 4: Ablation results on BFCL-v3. We show the effects of ablating out different components in our data synthesis pipeline. We also compare with different base models and different data sources. Results demonstrate the effectiveness of our training data from different aspects.

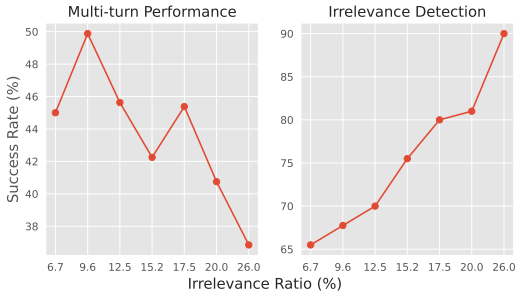

Figure 3: The performance when changing the data mixture with different number of irrelevance data.

section in Table 4, our MAGENT-7B-SFT surpasses other open-source data by a large margin, especially in the multi-turn scenario. We outperform Hammer2.1-7B by 3 points and models trained with APIGen and ToolAce data by 20.25. This demonstrates the effectiveness of our training data.

Base model We analyze on the effects of base model on the final performance. Besides the original Qwen2.5-Coder-instruct series, we compare with Qwen2.5-instruct series, which are trained without additional code data, and Mixtral-8x7B-instruct-v1. We observe that Coder series models, although obtaining slightly weaker performance on multi-turn and irrelevance detection without fine-tuning on our data, have better potential to learn from the training data, which achieves 5.38 better performance on multi-turn cases. Besides, by training comparing Mixtral-8x7B-instruct-v1 and MAGENT-Mixtral, we demonstrate that the performance boost brought by our data on function calling can be generalized to other models as well.

Discussion: the impact of data mixture We analyze the impact of data mixture to the final performance. As discussed in (Lin et al., 2024), the proportion of queries that involve missing or irrelevant functions would impact the overall behavior of models. We conduct an analysis to study the ratio of single-turn irrelevance samples versus the multi-turn samples. We fix the number of single-turn function call samples and multi-turn samples to 20k and 8k and adjust the ratio of irrelevance samples among 6.7%, 9.6%, 12.5%, 15.2%, 17.5%, 20.0%, and 26.3%, which corresponds to 2k, 3k ... 7k and 10k irrelevance samples. We test on our development set which consists of 200 irrelevance test cases and 200 multi-turn test cases. Figure 3 exhibits a performance trade-off between multi-turn success and irrelevance detection when adjusting the number of irrelevance examples. The optimal ratio of irrelevance data that balances the two aspects lies around 15% to 17%, based on which we set the final training data mixture in our case. The exact ratio is subject to changes based on different tasks and models but would provide a general guideline when considering data mixture.

5 Conclusion

We proposed a novel pipeline, MAGNET, for synthesizing multi-turn trajectories for training tool-use LLM agents. Targeted at the challenges in multi-turn FC, We proposed a graph-based multi-turn queries and reference FCs synthesis method to cover those challenges. We further converted those query-reference pairs into trajectories for both SFT and then mDPO training of LLMs. We demonstrated strong performance on agentic benchmarks.

Limitations

In this section, we discuss the limitations of the work. First, the function signatures we studied in the paper mainly consist of English and pure texts. It is possible some conclusions of this work might not generalize well to other languages and modalities. Future work could consider study multi-lingual and multi-modal tools as an extension to this work.

Second, in our qualitative study, we observe that our trained model might make mistakes when the knowledge retrieved by the tool is conflicted with the internal knowledge of the model. For example, consider a function `get_todays_date`, the tool might return a value that would be changing permanently. However, we found that even with the tool outputs, the model might still output some fixed date such as 2024-05-02. This reflects some limitations in resolving knowledge conflicts within context and internal knowledge.

Third, more exploration abilities could be incorporated into the model in future work. An ideal agent would be able to reflect on their wrong actions and restart the exploration, which is currently limited in our model, due to lack of such data in our training set.

References

Ibrahim Abdelaziz, Kinjal Basu, Mayank Agarwal, Sadhana Kumaravel, Matthew Stallone, Rameswar Panda, Yara Rizk, GP Shrivatsa Bhargav, Maxwell Crouse, Chulaka Gunasekara, et al. 2024. Granite-function calling model: Introducing function calling abilities via multi-task learning of granular tasks. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pages 1131–1139.

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Kinjal Basu, Ibrahim Abdelaziz, Kelsey Bradford, Maxwell Crouse, Kiran Kate, Sadhana Kumaravel, Saurabh Goyal, Asim Munawar, Yara Rizk, Xin Wang, et al. 2024. Nestful: A benchmark for evaluating llms on nested sequences of api calls. *arXiv preprint arXiv:2409.03797*.

Mingyang Chen, Haoze Sun, Tianpeng Li, Fan Yang, Hao Liang, Keer Lu, Bin Cui, Wentao Zhang, Zenan Zhou, and Weipeng Chen. 2024a. Facilitating multi-turn function calling for llms via compositional instruction tuning. *arXiv preprint arXiv:2410.12952*.

Yanfei Chen, Jinsung Yoon, Devendra Singh Sachan, Qingze Wang, Vincent Cohen-Addad, Mohammadhossein Bateni, Chen-Yu Lee, and Tomas Pfister. 2024b. Re-invoke: Tool invocation rewriting for zero-shot tool retrieval. *arXiv preprint arXiv:2408.01875*.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

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

Zhicheng Guo, Sijie Cheng, Hao Wang, Shihao Liang, Yujia Qin, Peng Li, Zhiyuan Liu, Maosong Sun, and Yang Liu. 2024. Stabletoolbench: Towards stable large-scale benchmarking on tool learning of large language models. *arXiv preprint arXiv:2403.07714*.

Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Diederik P Kingma. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Xian Li, Ping Yu, Chunting Zhou, Timo Schick, Omer Levy, Luke Zettlemoyer, Jason E Weston, and Mike Lewis. 2024. Self-alignment with instruction backtranslation. In *The Twelfth International Conference on Learning Representations*.

Qiqiang Lin, Muning Wen, Qiuying Peng, Guanyu Nie, Junwei Liao, Jun Wang, Xiaoyun Mo, Jiamu Zhou, Cheng Cheng, Yin Zhao, et al. 2024. Hammer: Robust function-calling for on-device language models via function masking. *arXiv preprint arXiv:2410.04587*.

Weiwen Liu, Xu Huang, Xingshan Zeng, Xinlong Hao, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Zhengying Liu, Yuanqing Yu, et al. 2024a. Toolace: Winning the points of llm function calling. *arXiv preprint arXiv:2409.00920*.

Zuxin Liu, Thai Quoc Hoang, Jianguo Zhang, Ming Zhu, Tian Lan, Shirley Kokane, Juntao Tan, Weiran Yao, Zhiwei Liu, Yihao Feng, Rithesh R N, Liangwei Yang, Silvio Savarese, Juan Carlos Niebles, Huan Wang, Shelby Heinecke, and Caiming Xiong. 2024b. APIGen: Automated PIpeline for generating verifiable and diverse function-calling datasets. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Jiarui Lu, Thomas Holleis, Yizhe Zhang, Bernhard Aumayer, Feng Nan, Felix Bai, Shuang Ma, Shen Ma, Mengyu Li, Guoli Yin, et al. 2024. Toolsandbox: A stateful, conversational, interactive evaluation benchmark for llm tool use capabilities. *arXiv preprint arXiv:2408.04682*.

Chang Ma, Junlei Zhang, Zhihao Zhu, Cheng Yang, Yujiu Yang, Yaohui Jin, Zhenzhong Lan, Lingpeng Kong, and Junxian He. 2024. Agentboard: An analytical evaluation board of multi-turn llm agents. *arXiv preprint arXiv:2401.13178*.

Thao Nguyen, Jeffrey Li, Sewoong Oh, Ludwig Schmidt, Jason Weston, Luke Zettlemoyer, and Xian Li. 2024. Better alignment with instruction back-and-forth translation. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 13289–13308.

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. 2023. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334*.

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 *ICLR 2024*.

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

Charlie Snell, Dan Klein, and Ruiqi Zhong. 2022. Learning by distilling context. *arXiv preprint arXiv:2209.15189*.

Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*.

Wei Xiong, Chengshuai Shi, Jiaming Shen, Aviv Rosenberg, Zhen Qin, Daniele Calandriello, Misha Khalman, Rishabh Joshi, Bilal Piot, Mohammad Saleh, et al. 2024. Building math agents with multi-turn iterative preference learning. *arXiv preprint arXiv:2409.02392*.

Fanjia Yan, Huanzhi Mao, Charlie Cheng-Jie Ji, Tianjun Zhang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. 2024. Berkeley function calling leaderboard.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. 2024. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115*.

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

Da Yin, Faeze Brahman, Abhilasha Ravichander, Khyathi Chandu, Kai-Wei Chang, Yejin Choi, and Bill Yuchen Lin. 2024. Agent lumos: Unified and modular training for open-source language agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12380–12403.

Jianguo Zhang, Tian Lan, Ming Zhu, Zuxin Liu, Thai Hoang, Shirley Kokane, Weiran Yao, Juntao Tan, Akshara Prabhakar, Haolin Chen, et al. 2024. xlam: A family of large action models to empower ai agent systems. *arXiv preprint arXiv:2409.03215*.

A Prompts

In this section, we list the prompts we used over the data synthesis and model inference process.

System prompts for training and evaluation We use the following system prompt following BFCL-v3 for both the training trajectories and the BFCL-v3, ToolQuery inference.

You are an expert in composing functions. You are given a question and a set of possible functions. Based on the question, you will need to make one or more function/tool calls to achieve the purpose. If none of the function can be used, point it out. If the given question lacks the parameters required by the function, also point it out. You should only return the function call in tools call sections. If you decide to invoke any of the function(s), you MUST put it in the format of [func_name1(params_name1=params_value1, params_name2=params_value2...), func_name2(params)]. You SHOULD NOT include any other text in the response. Here is a list of functions in JSON format that you can invoke.

For the list of functions, each function is formatted in this way:

```
template = {
    "category": "",
    "tool_name": "",
    "tool_description": "",
    "api_name": "",
    "api_description": "",
    "parameters": {
        "type": "dict",
        "properties": {
        },
        "required": [],
        "optional": [],
    }
}
```

Function domain classification prompt We use the following prompt to classify the domains of functions:

You will be given a few domains and a function from one of those domains. You will be given the function name, description, and the required parameters of it. Your task is to classify the function into one of the domains. The domains are: 'Cybersecurity', 'Artificial_Intelligence', 'Commerce', 'Advertising', 'Payments', 'News_Media', 'Cryptography', 'Devices', 'Business', 'eCommerce', 'Logistics', 'Finance', 'Events', 'Email', 'Business_Software', 'Music', 'Database', 'Translation', 'Jobs', 'Gaming', 'Monitoring', 'func_source_code', 'Education', 'Entertainment', 'Visual_Recognition', 'Sports', 'SMS', 'Media', 'Search', 'Finance', 'Location', 'Movies', 'Transportation', 'Text_Analysis', 'Mapping', 'Energy', 'Customized', 'Medical', 'Storage', 'Food', 'Health', 'Video_Images', "Science', 'Communication', 'Travel', 'Social', 'Data', 'Reward', 'Weather'. Return one line with the name of the domain. Or, if you cannot decide on which domain the function belongs to or think the function does not belong to any of the domains, output 'misc'.

Dependency prompt We use the following prompt to determine whether any of the candidates function could be neighbors to a target function:

You will be given a few API functions. You will also be given a target API. Your task is to create the adjacent list of the target API from those APIs. Each element in the adjacent list should be related to the target API. We say another function is related to the target API if: 1) the output of the target API is the premise of executing the function. For example, the output of fileexists('file.txt') API determines whether we can call downloadfile('file.txt'). 2) the output of the target API is exactly the input parameters of the function. For example, when calculating the area of a circle, the function getradius(obj) is the source node and calculate(radius) is the target node. 3) the output of the target API is partial input parameters of the function. For example, when posting something to social media, one might first get the content. In this case, the content = getcontent('file.txt') is the source node and posting(content, id, tags) is the target node. Notice that the relation

might cross the boundary of domains. For example, when the given APIs are in the domain of weather and travel, it is possible that a weather API could be related to a travel API since the weather determines the travel schedule. Also, the target API itself should not be in the adjacent list. For example, if the target API is get_id, there should not be a get_id function in the adjacent list. Return only the adjacency dictionary in a json format. Use exactly the original name of the tool as the key and values. In the adjacency dictionary, the only key is the target API, and each value is a list that contains the relevant APIs for that target API.

Check nested prompt We use the following prompt to determine whether two functions are nested:

You will be given two function information including their descriptions, parameters, response info etc. Your task is to determine whether the two functions can be nested. We call two functions to be nested when some parameter values for the later function call can be obtained by the first function call. For example when the first function is convert_usd_from_rmb(rmb_number=), and the second function is set_budget_limit(budget_limit_in_usd=). The two functions are nested because set_budget_limit needs a parameter value in dollars and convert_usd_from_rmb could output a dollar value. As another example, when the first function is get_airport_symbol_by_city(city=,range=), the second function get_flight_by_airport(airport_symbol=). The two functions are nested because the second function needs a symbol of airport while the first function provides that in the output. Please judge whether the input functions satisfy this nesting relationship. Return two lines: In the first line, If those two functions are nested, output yes, otherwise output no, Use lower case. In the second line, give a brief explanation on why you think they are nested.

Context distillation for positive trajectories prompt We use the following prompt for context distillation of positive trajectories: You are an expert in composing functions. You are given a question and a set of possible functions. Based on the question, you will need to make one or more function/tool calls to achieve the purpose. If none of the function can be used, point it out. If the given question lacks the parameters required by the function, also point it out. You should only return the function call in tools call sections. If you decide to invoke any of the function(s), you MUST put it in the format of [func_name1(params_name1=params_value1, params_name2=params_value2...), func_name2(params)]. You SHOULD NOT include any other text in the response. Here is a list of functions in JSON format that you can invoke. Notice that for each question, I already added hint function calls, following the [Hint] key words. Please compose your answer based on those hints while not mentioning those hints explicitly in your responses, i.e., when you decide to invoke function calls, just return the functions, and when you provide textual response, do not mention that there is a hint. Your textual response should summarize the function call outputs. Most of the time the hints are correct answers, just follow it... However, sometimes, those hints might not be perfectly correct, for example, you might see placeholders in the hints parameters like param1=unknow. So, when the hints are not correct, you need to identify them and compose the proper functions by looking for those parameter values from all previous turns. When you see [Hint]: miss function, this means the function needed in this step is missed. You should not simply output miss function in this case but try to use natural language to describe the situation and what functionality is missed. Similarly, when you see [Hint]: missed params, this means that some required parameters for the function is not mentioned in the query, just output some pure texts to ask for the

information. However, in your response, do not mention the hint, just answer to the query. When you encounter errors in function outputs, please try composing the functions again based on the error information in the errors. Do not just output textual response at once. **This is important**: when you see the [Hint] contains multiple function calls, i.e., more than one functions should be called for the query, this means those functions are relevant and nested. In this case, at each turn of your response, call only one function. Then, wait for the feedback from the user and then, call the next function. This is because sometimes the parameters of the later functions are missed without the user feedback. For example, when you see [Hint]: func_name1(params_name1=params_value1), func_name2(params_name2=params_value2), you should first output [func_name1(...)] with the correct parameter values and wait for the user response. Then, after you get the user response, based on the response, you call the next function [func_name2(...)] with the correct parameter values.

Hints selection for negative trajectories We use the following prompt for the judgement model which is also a Gemini-1.5-pro-002, for determining a negative trajectory hint: You will be given a multi-turn conversation between a user and an agent, the agent response for a single turn, which is possibly a function call, and a reference response. Your task is to judge whether the model response is a correct one based on the reference response. Below are possible error types. When both the reference and the model response are function calls, your judgement is for whether the model response accurately invoke the correct function call.

A response might be wrong in the following way:

1. Nested function calls: There are missing function calls. Model fails to call some necessary functions because they are not explicitly mentioned in the query.

2. Short dependency: There are outputs from a previous function call in this turn that is not used correctly in later function calls.

3. Long dependency: There are some parameter values exist in the conversation history but not properly used in this turn.

When both the reference and the model response are not function call but general textual response, your judgement is for whether the model response covers all the necessary information but also not hallucination based on the reference response.

4. Wrong summarization: whether the model response is a wrong summarization of the reference response.

When either one of the reference or the model response is not a function call while the other one is:

5. Missed function or parameters: there are some parameter values or functions present or not present in the context while the model thinks the opposite.

Additional guidelines: If one of the reference and model responses is function call while the other is not, directly output no.

Notice that when you see redundant parameters from the model response when it is function call, it might because it gives all the parameters even the default ones. So, as long as other parameters take the same values, regard this as correct.

In the first line, return yes or no. If your answer is no, in the second line, return a number to represent the error type.

Forth-translation prompt We use the following prompt for forth-translation to fill in function call parameters to make them executable: Now you are role-playing as a function-calling agent that involves in a multi-turn conversation with a user. You will be given the functions called by the history of this multi-turn conversation, indicated by round numbers. The functions called last round start with [Last Round].You will also be provided with a candidate function in a dictionary format with its descriptions and parameters. I would like you to generate the function

call for the next round using this function signature. Make sure the parameters for this candidate function should be derived from the user query and reference outputs from the last round function call. Rules: - You should use the function with the original name without any changes.
- For all the functions, make sure your generated function calls contain ALL the required parameters fields from the function documentation. You may also include some optional parameters. However, do not hallucinate any parameters outside of those. Use only the parameters indicated in the required and optional fields of the function documentation.
- Then, the parameter values for the new function should be related to the output from last round, please refer to the [Reference Output] for the corresponding values. - You can have parallel function call with the candidate function, i.e., call the function with different set of parameters, for your new query. However, **do not call more than three parallel functions**.
Format:
Thought: <the thought on which parameter values to use>
Answer: <You need to provide a groundtruth for the function calls that will be invoked in the next round as well as the parameters. Separate your reference function calls by comma. No any other separator is acceptable, only using comma. Also, if any of your parameters are with string value, use double quotation marks to include the parameters. If no answer can be generated, output FINISH in this line>

Back-translation prompt We use the following prompt for back-translation from a function signature to a query. The in-context examples are skipped for clarity:

Now you are role-playing as a user that involves in a multi-turn conversation with a function-calling agent. You will be given the functions called by the history of this multi-turn conversation, indicated by round numbers. The functions called last round start with [Last Round]. You will also be provided with a list of candidate functions in a dictionary format where the keys are the functions called last round and values are related and candidate functions that can be called in this round. I would like you to generate the query of this round which calls one or multiple functions from the candidate function list. When calling multiple functions, make sure you call no more than three functions at a single round.
Rules:
- The preferred next round query should be motivated by the outputs from the last round function output. Preferably, those outputs are used as the input parameters for as least one of the functions being called at this round.
- You should NOT mention which functions to use in your query explicitly.
- After you decide on which function to use, make sure your new query contains information for all the required parameters of the functions you want to call, although some information may be referred to implicitly as the outputs from the last round. If the value for some required parameters are not clear given the context, you may want to create a value for that required parameter but just remember, have information for all required parameters.
- Use no parameters besides the parameters indicated in the required and optional fields of the function documentation.
- For outputs from the last round, try not to mention the exact parameters that you will use. Instead, use references such as 'the location you just found', 'With the listed items'... to refer to the output of last round that will be leveraged next.
- Do not repeat any queries in the conversation history. This means your new query should not call the same function with the same set of parameters as any of the queries in the conversation, even the function exists in the adjacent list.
- Avoid using the APIs in [Do not use these APIs].
- Try to make the conversation as natural

```
as possible. Mind the logic between two
consecutive queries. Do not just create
an independent new query.
- Below are some examples of good output
given conversation history. Please follow
the style of conversation and make your
new query chained with previous queries.
```

B Training setup

We fine-tune Qwen2.5-Coder-7B-instruct and Qwen2.5-Coder-14B-instruct as the starting point and conduct SFT+RLHF over them. The reason for choosing these base models is that they have been adopted by other strong function calling models as the base model and have demonstrated strong potential for function calling abilities. All experiments are conducted on 16 Nvidia A100 GPUs on the same node. For SFT training, we fine-tune the full parameters for both sizes. We use a fixed max length of 8,172, warm up date of 0.1, Adam (Kingma, 2014) as optimizer and search over learning rate {1e-5, 5e-5}, batch size {64, 128} with gradient accumulation, and epochs {1, 2}. In general, we find that training for 1 epoch works the best. Other parameters are set as in the Section 4.1. For mDPO, we use LoRA tuning for 14B SFT model with a fixed rank 32 and alpha 64 and fully train the 7B SFT model. We search over learning rate {5e-7, 1e-6, 5e-6}, batch size {32, 64}, epoch {1, 2, 3}, beta {0.1, 0.01, 0.3}.

We use the transformer-trl [4] package for training SFT models and use the implementation from Xiong et al. (2024), which is also based on transformer-trl, for the mDPO training.

[4] https://github.com/huggingface/trl