Dynamic Tool Dependency Retrieval for Lightweight Function Calling

Bhrij Patel[*1,2], **Davide Belli**[*1], **Amir Jalalirad**[1], **Maximilian Arnold**[1], **Aleksandr Ermolov**[1], **Bence Major**[1]

[1]Qualcomm AI Research
[2]University of Maryland, College Park, USA

Correspondence: bbp13@umd.edu, {dbelli, ajalalir, marnold, aermolov, bence}@qti.qualcomm.com

Abstract

Function calling agents powered by Large Language Models (LLMs) select external tools to automate complex tasks. On-device agents typically use a retrieval module to select relevant tools, improving performance and reducing context length. However, existing retrieval methods rely on static and limited inputs, failing to capture multi-step tool dependencies and evolving task context. This limitation often introduces irrelevant tools that mislead the agent, degrading efficiency and accuracy. We propose Dynamic Tool Dependency Retrieval (DTDR), a lightweight retrieval method that conditions on both the initial query and the evolving tool calling plan. DTDR models tool dependencies from function calling demonstrations, enabling adaptive retrieval as plans unfold. We benchmark DTDR against state-of-the-art retrieval methods across multiple datasets and LLM backbones, evaluating retrieval precision, downstream task accuracy, and computational efficiency. Additionally, we explore strategies to integrate retrieved tools into prompts. Our results show that DTDR improves function calling success rates between 23% and 104% compared to state-of-the-art static retrievers.

1 Introduction

Large language models (LLMs) augmented with tool use (a.k.a., function calling) have rapidly evolved from early neuro-symbolic systems to agentic frameworks that can plan, select, and invoke external Application Programming Interfaces (APIs) (Yao et al., 2023; Schick et al., 2023; Patil et al., 2024; Patel et al., 2025). Despite this progress, deploying tool-augmented LLMs on-device remains challenging due to two key constraints: (i) *lighweight* under strict memory and latency budgets, and (ii) *effectiveness* across large and heterogeneous tool sets.

Therefore, prior work has proposed using tool *retrieval* modules to encode only relevant tools into the prompt of the function calling agent (Qin et al., 2023; Braunschweiler et al., 2025; Paramanayakam et al., 2025). Doing so makes it easier for the agent to identify the correct function, reducing unnecessary calls, and enhancing both accuracy and prompt efficiency. However, a major challenge is determining what information should guide the tool retrieval. Some methods rely solely on semantic similarity between the query and tool descriptions (Gao et al., 2025; Paramanayakam et al., 2025), which can overlook the history of selected tools in the ongoing, multi-step plan. Others leverage static tool dependency graphs built from demonstration trajectories (Liu et al., 2024a), but these approaches risk retrieving tools irrelevant to the query or being biased toward repeated calls.

An effective tool retriever must adapt to both the *current task* and the *ongoing trajectory*, while remaining lightweight enough to satisfy on-device constraints. This setting raises the question: can such specificity be achieved with a low-resource approach? To answer this, we introduce **Dynamic Tool Dependency Retrieval (DTDR)**, a lightweight retrieval component that, given a user query and partial plan, identifies a small set of relevant tools and their dependency relations (see Figure 1). Our main contributions are as follows:

1. **Lightweight Tool Dependency Retrieval method.** We formulate *Dynamic Tool Dependency Retrieval* (DTDR), a dependency-aware tool retrieval framework that conditions on both the user query and the evolving tool plan to recover a minimal, task-specific dependency subgraph. Through comprehensive analysis, we provide strong evidence that methods without query- and history-awareness are unable to solve the tool retrieval task, so history-aware methods should be adopted.

[*] Equal Contribution. [1] Qualcomm AI Research is an initiative of Qualcomm Technologies, Inc. [2] Work done during an internship at Qualcomm AI Research, Amsterdam.

Findings of the Association for Computational Linguistics: ACL 2026, pages 33649–33672

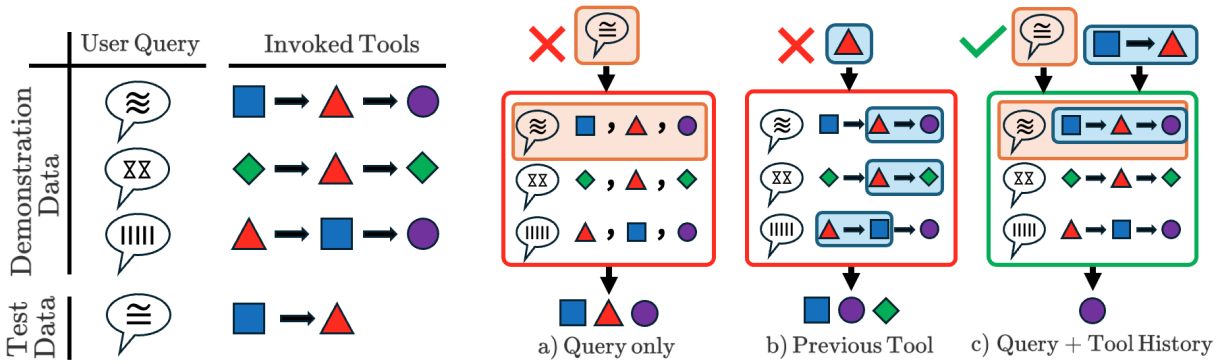

Figure 1: **Dynamic Tool Dependency Retrieval.** Given demonstration data for a set of tools, previous work retrieves tools based on either a) the natural language query (highlighted in orange) or b) the latest executed tool call in the plan (the red triangle, highlighted in teal). We instead propose a retrieval method that is conditioned on both the query and the growing history of tool calls (blue square, then red triangle). This method allows for retrieving only tools that are strictly relevant to both query (orange) and tool context (teal).

2. **Extensive Retrieval and End-to-End evaluation.** We conduct a systematic comparison against a suite of retrieval baselines (text-based, embeddings-based, and dependency/graph-based), showing that our dynamic variant outperforms previous work in terms of *retrieval* metrics (MRR/F_1-score), *downstream* performance (function selection accuracy and end-to-end task success), and *efficiency* (footprint, token budget) across several datasets and LLM backbones of varying sizes.

3. **Analysis with multiple Prompting strategies.** We use a prompt-efficient in-context learning (ICL) representation that conditions the LLM only on the minimal subgraph of tool dependencies retrieved. We benchmark multiple ICL encoding strategies, identifying weighted Hard Masking as the best contender, but also investigating when and why other approaches should be preferred based on different factors (model scale, dataset statistics, tool retrieval accuracy).

2 Related Work

Recent LLMs demonstrate impressive tool-usage capabilities, with cloud-based models such as GPT-5 (OpenAI, 2025), Claude 4 (Anthropic, 2025), and GLM-4.5 (Zeng et al., 2025) leading by a wide margin on function-call benchmarks like BFCL V4 (Patil et al., 2025) and τ^2-Bench (Barres et al., 2025). Recent work has explored fine-tuning LLMs on large datasets to build general-purpose function calling models (Cheng-Jie Ji et al., 2024). These models often fail in real-world scenarios due to poor function selection, misinterpretation of user intent, or under realistic data perturbations (Dang et al., 2025; Rabinovich and Anaby-Tavor, 2025). Alternatively, ICL strategies for tool learning can involve including tool descriptions (Shen et al., 2023, 2024; Patel et al., 2025) or example trajectories (Paranjape et al., 2023; Sarukkai et al., 2025) within the model prompt. To handle hundreds of functions, recent work uses retrieval-augmented generation to sub-select tools that are relevant to the task (Qin et al., 2023; Braunschweiler et al., 2025; Paramanayakam et al., 2025). Most prior art retrieves relevant tools based on the query and tool descriptions (Braunschweiler et al., 2025; Paramanayakam et al., 2025; Paranjape et al., 2023). (Liu et al., 2024b) and (Ding et al., 2025) assume a known tool dependency graph and use it to aid the LLM with selecting relevant functions. As tool dependencies are often unknown, others propose learning them via demonstrations (Paranjape et al., 2023; Chen et al., 2025; Liu et al., 2024a; Qin et al., 2023; Patil et al., 2024; Erdogan et al., 2024). Unlike prior retrievers that rely solely on the user query or static tool dependency graphs, we propose a dynamic tool retrieval approach that conditions on both the current query and the evolving sequence of previously invoked tools. In Table 1, we summarize related works that most closely align with our method. We compare against 5 different categories. **1) Tool Dependency Aware:** retrievers that utilize tool dependencies are better informed to solve multi-step tasks (Gao et al., 2025); **2) Tool Description Free:** function documentation may not be available or consistent across large sets of tools (Patel et al., 2025); **3) Query Aware:** retrieval methods should retrieve tools relevant to the specific task, i.e. email tasks will more likely

Method	Tool Deps. Aware	Tool Desc. Free	Query Aware	Multi-Step History Aware	Small model
BM25 (Robertson et al., 2009)	✗	✗	✓	✗	✓
ToolGraph Retriever (Gao et al., 2025)	✓	✗	✓	✗	✗
Less-is-More Lv1 (Paramanayakam et al., 2025)	✗	✗	✓	✗	✓
ToolNet (Liu et al., 2024a)	✓	✓	✗	✗	✓
TinyAgent (Erdogan et al., 2024)	✗	✓	✓	✗	✓
Toolformer (Schick et al., 2023)	✗	✓	✓	✗	✓
DTDR (Ours)	✓	✓	✓	✓	✓

Table 1: Comparison of tool retrieval methods used in prior work. Our work is the only one that satisfies all 5 desired conditions.

need getting an email address than opening a document, **4) Multi-Step History Aware:** retrievers that take into account multi-step history, not just current step, avoid bias towards frequently called functions, and **5) Small Model:** On-device agents have stricter memory and latency requirements that the retrieval method needs to satisfy. Our method is the only one that satisfies all 5 categories. A more extensive discussion of related works can be found in Appendix A.

3 Problem Formulation

Retrieval-Augmented Function Selection. Consider the function selection agent π, consisting of a frozen language model (LM). Let $\mathcal{F}$ be the collection of function names that the agent can choose from to solve a task described through a natural language query q, such as *"Reply to my latest email from Willem."*. Let $f_{0:t-1} = [f_0, f_1, ..., f_{t-1}]$ be the list of previous functions predicted for q up until timestep t. Typically, q, $f_{0:t-1}$ and $\mathcal{F}$ are encoded in the LM prompt, p, which is input to the agent to sample a function $f_t \sim \pi(\cdot|p)$. A sampled function f_t is correct if $f_t \in \mathcal{F}_t^*$, the set of correct functions for q at t. To raise the probability of said correctness, prior work encodes into the prompt a subset of $\mathcal{F}$ (Qin et al., 2023; Braunschweiler et al., 2025). This subset is retrieved via some module $\omega(\cdot)$, a function that outputs a subset of function names $\mathcal{F}_t \subseteq \mathcal{F}$. The set of inputs to the ω is dependent on the specific retrieval method. Let T be the maximum trajectory length for any q, let Ω be the set of all possible ω retriever functions, and let $\mathbf{1}$ be a scalar indicator function. Our function selection problem is a prompt optimization objective:

$$\max_{\omega \in \Omega} \sum_{q \in Q} \sum_{t=0}^{T} \mathbf{E}_{f_t \sim \pi(\cdot|p)}[\mathbf{1}_{f_t \in \mathcal{F}_t^*}]. \quad (1)$$

This optimization problem mathematically grounds the success of the function calling agent π to how well the retrieved set $\mathcal{F}_t$ aligns with $\mathcal{F}_t^*$. Specifically, we consider an optimal retrieval module ω^* as one such that it retrieves a non-empty subset of the ground-truth set: $\{\} \subset \omega^*(\cdot) = \mathcal{F}_t \subseteq \mathcal{F}_t^*$. Note that we focus on optimizing the individual function selection step for more fine-grained downstream evaluation of the retrieval module, so this objective function is less sparse than ones for multi-step, function-calling tasks (Patel et al., 2025). We discuss in Section 6 end-to-end results with trajectory-level evaluation.

In our setting, the agent does the entire planning (retrieval and function selection) without any function calling interleaved. Our scenario for tool selection remains a valid and practical setting for agentic pipelines. In many real-world systems—such as AI-based IDE assistants (Google, 2025), workflow automation agents, or enterprise copilots—the agent first generates a plan and only then executes the selected tools. In these settings, accurate tool selection at planning time is critical: incorrect planning can lead to unnecessary execution and costly re-planning. Therefore, improving tool selection in an execution-free, offline setting directly strengthens the planning phase of agentic systems and increases efficiency and latency predictability before any tool invocation occurs.

Limitation of Prior Work: Suboptimal Sets of Inputs. Some zero-shot retrieval methods (Paramanayakam et al., 2025; Gao et al., 2025) implement ω as the cosine similarity between embeddings of the query (or a transformation of it) and embeddings of each tool description. Furthermore, the retrieved set $\mathcal{F}_t$ is used throughout the entire trajectory, constant throughout the T timesteps. Relying solely on static tool descrip-

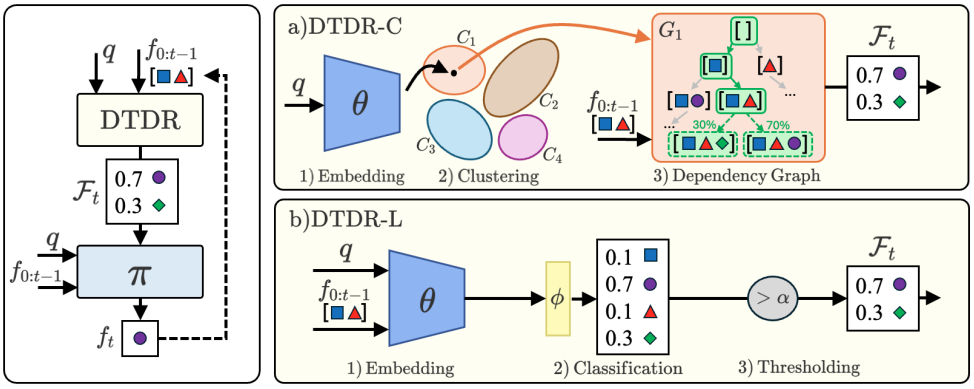

Figure 2: **System diagram for DTDR**. On the left, the user query and tool history are input to DTDR to retrieve the most likely next tools. The LLM π selects the next tool among this set. On the right, we show the two alternative instantiations for the retriever: a) DTDR-C, based on a clustering step to retrieve an explicit graph of tool dependencies; and b) DTDR-L, based on a learned linear classifier implicitly modeling tool dependencies. Both systems are conditioned on user query and full tool history.

tions often overlook the dynamic nature of tool usage in context. While semantic similarity between queries and tool descriptions may capture surface-level relevance (Lin et al., 2024), it fails to account for how tools are actually sequenced in real tasks. Other work such as (Liu et al., 2024a) utilize collected demonstration trajectory data D of function calling tasks, obtained by tool providers (Paranjape et al., 2023), historical tool trajectories (Chen et al., 2025) or generated (Erdogan et al., 2024; Gao et al., 2025; Patil et al., 2024). With D, they compute empirical next-function probabilities based on the latest function f_{t-1} to obtain $\mathcal{F}_t$. However, this limited context does not model multi-step tool dependencies well. For example, if sending the same email to multiple people, *getEmailAddress* must be called consecutively multiple times. Demonstrations that contain function trajectories such as these will cause a high value of $P(\textit{getEmailAddress}|\textit{getEmailAddress})$, thus biasing the agent to repeatedly call it more times than necessary. Furthermore, computing probabilities based on the entire demonstration data across various queries underfits tool-use patterns. The $P(\textit{getEmailAddress}|\textit{getEmailAddress})$ value suitable in tasks with multiple recipients can be misleading in tasks with a single recipient. More generally, conditioning only on the most recent function fails to capture branching dependencies that depend jointly on earlier steps and task intent. For instance, after calling *open_file* followed by *summarize_pdf*, the appropriate next action differs depending on whether the task requires appending notes, composing an email, or performing another file operation. A first-order model aggregates all continuations of *summarize_pdf* across diverse tasks, assigning non-zero probabilities to tools that are valid in other queries but not appropriate in the current plan. This misalignment results in an overly broad candidate set that lacks task-specific precision. Without incorporating task-specific or higher-order information, the context of previous functions can therefore act as a misleading prior for π Without the task-specific information, the context of previous functions may be a misleading prior for π. We aim for a *lightweight retrieval mechanism to obtain tool dependency priors that capture both task-specific and context-specific information*.

4 Proposed Approach: Dynamic Tool Dependency Retrieval

To capture appropriate tool dependencies for Retrieval-Augmented Function Calling, we introduce *Dynamic Tool Dependency Retrieval (DTDR)*. The high-level idea of DTDR is simple: the retrieval module ω should be based on both the task q and the trajectory of previous function calls $f_{0:t-1}$; thus, $\omega(q, f_{0:t-1})$. Figure 2 (left) gives an overview of our approach. The test query q and the trajectory of function calls $f_{0:t-1}$ are input to DTDR, which predicts a set of tools $\mathcal{F}_t$, assigning a probability to each. We then encode the retrieved dependencies into the prompt p by hard masking the full set of tools $\mathcal{F}$, while also providing the probabilities for the retrieved tools. In Section 5, we elaborate on other ICL methods we investigated to embed $\mathcal{F}_t$ into the prompt. The encoded prompt is then processed by the LLM agent to sample the next

function $f_t \sim \pi(\cdot|p)$. For end-to-end evaluation, this continues in an iterative approach starting from the empty function call plan, and ending after predicting the "end-of-plan" function.

Variants of DTDR. We propose two lightweight variants for DTDR: one supervised gradient-based method and one unsupervised clustering-based method. Firstly, proposing two allows us to validate whether our claims on *dynamic* tool retrieval are consistent across different categories of methods. Secondly, it enables us to directly compare against previous work belonging to these two categories. Lastly, depending on software and system constraints, either variant might be preferred for a real on-device scenario. Both variants utilize demonstration data $\mathcal{D}$.

Dynamic Tool Dependency Retrieval-Clustering (DTDR-C). This variant (see Figure 2, top-right) is based on a clustering component and a graph-traversal component. The clustering component maps a test query to a tool dependency graph which is relevant for the specific task. The graph-traversal component traverses the graph based on the history of tool calls, and determines the most likely next tools for the plan. Formally, we embed demonstration queries $Q_{\mathcal{D}}$ with a pretrained embedding model and fit a K-Means clustering model C on the embeddings. Let β be the embedding of q and d be the ground-truth demonstration data for q. For each cluster index k, we consider the set of demonstrations assigned to that cluster $D_k = \{d \mid \forall q \in Q_{\mathcal{D}} \; s.t. \; C(\beta) = k\}$. We build a weighted tool dependency graph G_k describing the next-tool dependency probabilities given a sequence of tools. In particular, $G_k(f_{0:t-1})$ describes the set of functions which follow the history $f_{0:t-1}$ in the demonstration data D_k, as well as their probabilities. Details on constructing the tool dependency graph G are included in Appendix C and Appendix I. Given a test query q and history $f_{0:t-1}$, the next-tool prediction in DTDR-C can be obtained with $\mathcal{F}_t = G_{C(\beta)}(f_{0:t-1})$. The number of learned parameters in the K-means model is $e * K$, with e being the embedding dimensionality and K the number of clusters.

Dynamic Tool Dependency Retrieval-Linear (DTDR-L). This supervised learning variant (see Figure 2, bottom-right) is based on a linear layer classifier trained to predict the set of next functions given the test query and the history of previous tool calls. We train a 1-linear-layer classifier ϕ on top of a frozen embedding model to predict the next function given both the query and the function trajectory until t. Let ζ be the embedding of $q + f_{0:t-1}$, where the operator $+$ denotes concatenation between strings. We use a softmax operation to obtain a probability distribution from the model outputs. Thus $\phi(\zeta)$ is the probability of f given the query and ongoing, current trajectory. To retrieve only a subset of tools, we set a threshold value α, so that $\mathcal{F}_t = \{f | f \in \mathcal{F}, \phi(\zeta) > \alpha\}$. The number of learned parameters in this model is $e * |\mathcal{F}|$, only depending on the output size of the embedding model e and number of tools $|\mathcal{F}|$. Additional implementation details and pseudocode can be found in Appendix C and Appendix I.

5 Experimental Setup

Datasets. We benchmark on four function calling datasets: *TinyAgent* (Erdogan et al., 2024), *TaskBench DailyLife APIs*, *TaskBench HuggingFace*, and *TaskBench Multimedia* (Shen et al., 2024). The first two datasets represent tools commonly available on a typical device, the other two datasets are specific to a particular domain, which makes them more challenging for "out-of-the-box" LLMs. We report in Appendix B the statistics for each benchmark, with an important insight being that all datasets have between 1 and 2 tool dependencies per plan, except for TaskBench DailyLife APIs which has almost zero dependencies. For each dataset, we set aside around 30% of the data for testing, while the rest is used as demonstration trajectories for tool retrieval and ICL methods.

LLM backbones. We evaluate tool-retrieval and ICL methods applied to 7 LLM backbones. We consider the *Qwen 3* family of models (0.6B, 1.7B, 4B, 8B, 14B) (Yang et al., 2025) representing edge devices of varying size, *GPT-4o* (Hurst et al., 2024) representing a cloud model solution, and *Gorilla-V2* (Cheng-Jie Ji et al., 2024) representing an edge-device fine-tuned on function calling data. Assuming INT4 quantization and a KV cache of up to 10 thousand tokens, Qwen 3 models up to Qwen 3 4B could efficiently run on a typical mobile device (Federici et al., 2025; Song et al., 2025).

Metrics. For retrieval performance, we measure *Mean Reciprocal Rank (MRR)* and F_1 score. To analyze downstream performance, we look at both *Function Selection Accuracy (FSA)* and *Success*

Retrieval Method	TinyAgent			Taskbench-HF			Taskbench-MM		
	FSA	MRR	F_1	FSA	MRR	F_1	FSA	MRR	F_1
Random Guess	5.9	0.26	0.12	4.2	0.16	0.05	2.4	0.11	0.03
BM-25 (Robertson et al., 2009)	23.1	0.35	0.20	8.3	0.18	0.05	13.7	0.26	0.19
QTS (Vanilla)	15.8	0.26	0.14	7.9	0.21	0.08	5.3	0.14	0.04
QTS (Gao et al., 2025)	21.5	0.18	0.09	14.6	0.37	0.27	5.3	0.14	0.20
QTS (Paramanayakam et al., 2025)	23.7	0.36	0.19	12.9	0.30	0.18	7.6	0.29	0.18
DR (Liu et al., 2024a)	30.7	0.70	0.49	17.1	0.46	0.33	13.3	0.38	0.27
Dynamic DR (DTDR-C) (Ours)	43.3	0.78	**0.56**	27.5	0.64	0.55	24.4	0.52	0.43
LR (Erdogan et al., 2024)	25.6	0.53	0.39	20.5	0.53	0.32	21.0	0.49	0.28
Dynamic LR (DTDR-L) (Ours)	**65.1**	**0.93**	0.55	**27.8**	**0.75**	**0.63**	**27.0**	**0.69**	**0.55**

Table 2: Retrieval performance for methods from different categories (QTS = Query-Tool Similarity, LR = Learned Retriever, DR = Dependency Retriever). QTS and LR categories are query-conditioned, while DR is history-conditioned. Our Dynamic methods are conditioned on both query and tool history, which yields improved performance. Function Selection Accuracy is reported on Qwen 3 0.6B using hard masking as ICL method. Best results per dataset and metrics are in bold.

Rate (SR) for the end-to-end task, which considers both function selection **and** parameter filling. Full details in Appendix D. We assess efficiency via *prompt length* as a proxy for prefill speed (or time to encode and process the whole prompt); and *number of parameters* to quantify the LLM footprint which determines on-device usability.

Tool Retrieval Baselines. We consider different categories of Tool Retrievals baselines from recent work presented in Table 1. Classical term-similarity baselines like BM-25 (Robertson et al., 2009) directly compare text similarity between documents. *Query-Tool Similarity (QTS)* baselines use pre-trained sentence-embedding models to encode descriptions of queries and tools, without learning from demonstrations. This approach is extended in Less-is-More Level 1 (Paramanayakam et al., 2025) by prompting an LLM to describe the ideal tool set to solve a query, and in Tool Graph Retriever (Gao et al., 2025) by additionally encoding tool dependencies in the embeddings. *Learned Retriever (LR)* baselines as in Erdogan et al. (2024) fine-tune small classifiers to learn what functions are useful to resolve a query. Finally, *Dependency Retriever (DR)* methods like ToolNet (Liu et al., 2024a) leverage tool dependencies to directly sub-select viable tools based on the most recent tool in the plan. DTDR introduces *dynamic* retrieval methods that leverage both the *query* and evolving *tool-call history* to improve tool prediction. These are referred to as a *Dynamic Learned Retriever* (DTDR-L) and a *Dynamic Dependency Retriever* (DTDR-C). As with DTDR, our baselines do not assume execution feedback like works such as SEER (Cui et al., 2025). Additional details for baselines and methods in Appendix C.

ICL Methods for Encoding $\mathcal{F}_t$ into Prompt. For all methods, including the *No ICL* baseline, we provide 1 randomly selected demonstration to show how to complete an example task. For all other ICL methods, we encode the retrieved tools. For the *Raw Demonstrations* method, we provide up to 5 additional demonstrations for which the plan includes the predicted tools. For more efficient prompting, we can either use the *Hard Mask* and completely exclude the remaining tools from the prompt, or a *Soft Mask*, where the full set of tools is presented to the model but the retrieved list is emphasized as the set the model should generally prefer. Since retrieval methods provide scores for each tool, these can be included encoded, yielding *Weighted* variants of the two masking approaches. We provide example ICL prompts in Appendix G.

6 Results and Discussion

Does DTDR improve tool retrieval? We evaluate Tool Retrievers on ranking, retrieval and downstream performance (see Table 2). MRR scores generally correlate well with Function Selection Accuracy, indicating that better ranking improves LLM reasoning. Set-based metrics like F_1 score are less informative of the downstream performance, as they do not capture the relative importance of tools. Overall, *Learned Retriever* and *Dependency Retriever* perform better than all baselines. Our *dynamic* Learned Retriever (DTDR-L) surpasses the *static* counterpart (Erdogan et al., 2024) by over 35% on TinyAgent and TaskBench DL, while

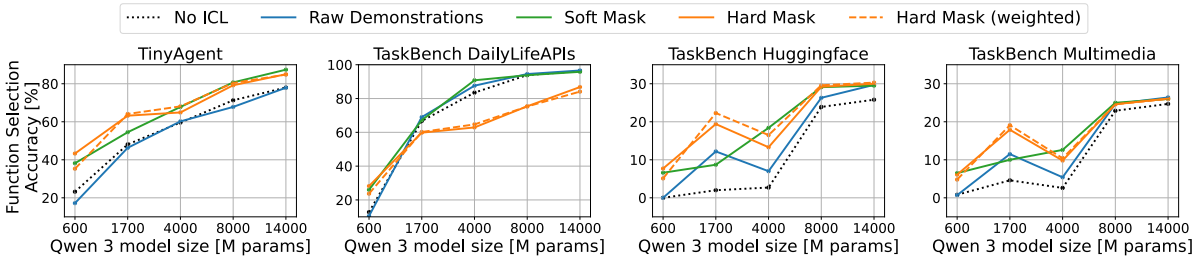

Figure 3: Comparison of efficient ICL methods against ICL with Raw Demonstrations and the baseline without ICL. All results are conditioned using the DTDR-C retriever. Pruning irrelevant tools from the prompt has a greater impact for smaller models that cannot handle longer contexts.

Query: *Summarize the content of "ResearchPaper.pdf" and append the summary to the "Research Notes" in the "Academic" folder. Then, compose an email with the subject "Research Summary" and attach the PDF file to send to your professor.*
Current tool plan: [*open_file, summarize_pdf*]
Ground Truth Set of next tools: [*append_note_content, get_email_address*]

Figure 4: Example tool selection step in TinyAgent.

matching performance on TaskBench HF and MM. This demonstrates the benefit of conditioning the prediction on the function selection history. Similarly, DTDR-C improves over the *static* Dependency Retriever counterpart (Liu et al., 2024a) by 50–100%, confirming the value of leveraging query and full history for retrieval, instead of only considering the latest function in the plan. For a qualitative analysis, we compare tools retrieved from DR, LR, and DTDR-L for the example in Figure 4. For DR, the last tool call in the history (*summarize_pdf*) is the only input for this tool retriever. As *summarize_pdf* appears in queries with different contexts in the demonstration data, DR predicts a high probability (between 5% and 20%) for 7 tools. Because many TinyAgent queries involving PDFs require opening and summarizing files, LR (query-aware only) assigns high probability to [*summarize_pdf, open_file, end_plan*], even though these tools have already been selected in the current plan. DTDR-L retrieves [*append_note_content, get_email_address, compose_email*].

How to encode tools in prompt? In Figure 3 we compare different ways to represent retrieved tools in the prompt. In three of four datasets, Hard and Soft Masking consistently outperform both the No ICL and the Raw Demonstration baselines. The exception is TaskBench DL, which lacks intrinsic function dependencies (see Table 5), making dependency-based prompting less relevant. Hard Masking achieves the highest accuracy for smaller models by directly pruning the set of available tools presented to the LLM. Weighted Hard Masking performs comparably to its unweighted variant. Despite similar overall performance, we observe that weighted masking results in better accuracy when the retriever assigns high probability to the correct tools, and unweighted masking is better in the opposite scenarios. This suggests that unweighted masking should be preferred when expecting a large distribution shift between test queries and demonstration data, as predicted probabilities will be less accurate (details in Appendix F).

How does DTDR perform on different datasets and model size? In Table 3, we evaluate the Function Selection Accuracy (FSA) of best tool retrievers on several datasets and models of different sizes. Results on additional models and datasets can be found in Table 8 in the Appendix. DTDR-C consistently outperforms its static DR counterpart across all settings, while DTDR-L is comparable or better than static LR. This supports our hypothesis that dynamic retrieval is key for high-quality function calling. Among all methods, DTDR-L achieves the best overall performance. Notably, DTDR-L applied to Qwen 3 4B and 8B respectively surpasses the No ICL baseline applied to Qwen 3 14B and GPT-4o, bridging the performance gap between edge and cloud models. In Figure 3, the Raw Demonstrations approach hurts smaller models but become competitive at larger scales, where LLMs exhibit stronger reasoning capabilities. However, including raw demonstrations in the prompt increases its length, which results in longer prefill time and worse compute efficiency. We analyze this in Figure 5, where we compare the prompt lengths obtained with different retrieval

	Method	TinyAgent		TB-DL		TB-HF		TB-MM	
		FSA (↑)	SR (↑)	FSA (↑)	SR (↑)	FSA (↑)	SR (↑)	FSA (↑)	SR (↑)
Qwen3 0.6B	No ICL	23.2	0.4	12.8	0.0	5.5	0.0	7.3	0.0
	DR (Liu et al., 2024a)	22.0	0.2	11.5	0.0	6.7	0.0	9.5	0.3
	LR (Erdogan et al., 2024)	19.8	0.9	19.6	2.0	10.0	0.5	12.1	0.7
	DTDR-C (Ours)	35.3	**4.2**	23.7	**6.8**	**20.8**	**5.9**	**20.2**	**8.3**
	DTDR-L (Ours)	**46.1**	3.5	**45.9**	0.8	18.9	1.2	17.6	3.7
Qwen3 1.7B	No ICL	48.1	4.4	66.7	13.5	32.2	3.6	38.1	4.9
	DR (Liu et al., 2024a)	51.6	2.6	60.0	9.7	41.6	5.7	45.1	6.8
	LR (Erdogan et al., 2024)	50.5	7.1	56.6	20.3	42.8	17.3	42.1	17.3
	DTDR-C (Ours)	64.1	11.8	60.2	26.4	**52.8**	**21.3**	**47.8**	**22.5**
	DTDR-L (Ours)	**78.4**	**14.5**	**83.1**	**29.4**	49.0	19.7	45.5	20.4
Qwen3 4B	No ICL	59.6	5.0	83.5	13.5	45.8	5.1	54.3	6.8
	DR (Liu et al., 2024a)	62.7	9.2	64.7	19.3	52.8	19.3	51.8	16.8
	LR (Erdogan et al., 2024)	52.2	9.0	60.4	19.3	56.9	28.0	58.2	26.5
	DTDR-C (Ours)	68.1	14.4	64.7	27.9	56.0	**35.1**	52.4	28.0
	DTDR-L (Ours)	**80.7**	**16.9**	**89.0**	**31.8**	**60.5**	29.8	**64.1**	**30.9**

Table 3: Comparison of selected methods in terms of function selection accuracy and end-to-end success rate over different datasets and models. All retrievers are paired with weighted Hard Masking for ICL. Best results per dataset and model are bolded. DTDR reaches higher accuracy as it retrieves a set of tools more closely aligned with $\mathcal{F}_t^*$.

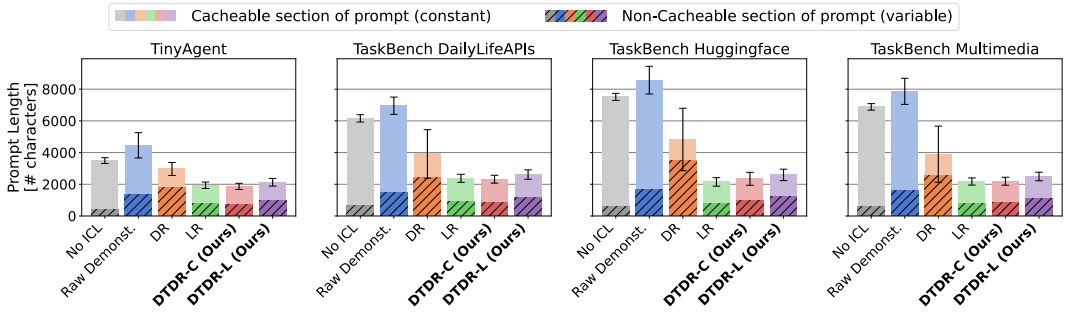

Figure 5: Prompt length across different methods and datasets. Our method reduces the prompt length by: 1) efficiently encoding ICL examples as tool dependencies instead of Raw Demonstrations, and 2) only retrieving tool dependencies which are relevant for the test query.

methods, distinguishing between constant and variable prompt sections. The constant section (guidelines, task demonstrations, and tool descriptions) can be precomputed and cached, while the variable section (query encoding and tool descriptions) must be recomputed for each query (see examples in Appendix G). Instead of Raw Demonstrations, we can directly use the retriever outputs to mask the set of tools provided to the LLM, which drastically reduces the prompt size. Static DR reduces the prompt length by selecting a subset of functions, but the subset changes at each step and must be updated dynamically in the prompt. Static LR can select a narrower set of tools, but it tends to overfit to retrieving the most frequent ones, which results in lower end-to-end performance. DTDR-C and DTDR-L can both reduce the size of the retrieved tool set and retain good downstream accuracy. In particular, they decrease the total prompt length by up to 73% and the variable portion by up to 48% when compared to raw text demonstrations. In terms of compute costs, DTDR and most baseline retrievers use a small sentence encoder to represent the query, whose cost is negligible relative to the subsequent LLM prompting step. The only exception incurring overhead at inference time is the QTS retriever proposed by Paramanayakam et al. (2025), which additionally prompts an LLM to describe the ideal tool set for the task.

Does DTDR improve end-to-end function calling success rates? In Table 3 we also report end-to-end Success Rate (SR) for the function calling task. We use the same model with different prompts for the function selection and parameter filling subtasks (details in Appendix G). If one or more mistakes occurs at any step in the plan (either in the

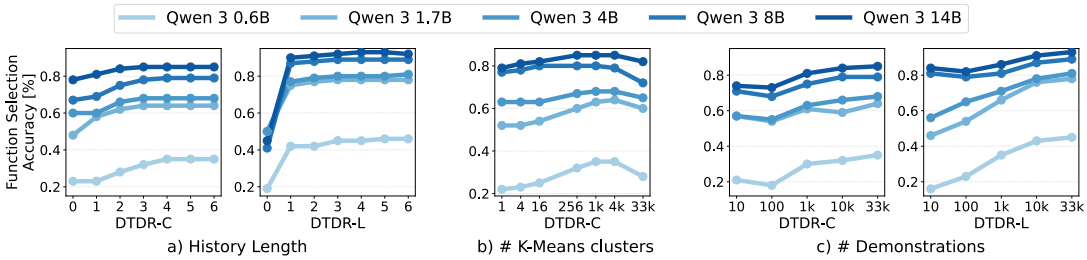

Figure 6: Ablations on: a) history length, b) # of k-means clusters, and c) # of demonstrations.

function name or its parameters), the whole plan is evaluated as unsuccessful. For datasets with tool dependencies (TA, TB-HF, TB-MM), the dynamic variants of DTDR improve the end-to-end Success Rate between 300% and 600% compared to the baseline without ICL, and between 15% and 200% compared to the best Tool Retrieval baseline. Weaker baselines on smaller models fail on all datasets. LR (Erdogan et al., 2024) tends to overfit to predicting the most common function in the dataset, while the same architecture in its dynamic variant DTDR-L (adding function call history as input), prevents this problem and yields significantly better performance. Using Raw Demonstrations yields good performance with larger models, as they have increased capabilities for longer context. However, as previously discussed, this approach would yield significant increases in prefill latency.

Impact of History Length, Clustering, and Number of Demonstrations. Figure 6a examines the effect of varying the history length l in the function call sequence $[f_{t-l}, ..., f_{t-1}]$. Longer histories enhance dependency modeling in DTDR-C and provide richer context for DTDR-L. Both models benefit up to $l = 3$, beyond which gains taper off. Without history, DTDR-L tends to overfit, often defaulting to the most frequent functions, as previously observed. Figure 6b explores query-conditioning by varying the number of clusters in DTDR-C. With a single cluster, retrieval is query-agnostic, similar to (Liu et al., 2024a). Increasing the number of clusters enables more targeted retrieval from demonstrations similar to the test query. The extreme case (nearest-neighbor retrieval) assigns one demonstration per cluster. We find optimal performance when the number of clusters is approximately $1/10$ of the demonstrations, which we adopt as the default setting for all DTDR-C experiments. Figure 6c shows how increasing the number of demonstrations influences Function Selection Accuracy for both DTDR-C and DTDR-L.

	Plan Length							
Method	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
DR	44.6	36.5	45.1	53.7	62.7	61.6	51.7	50.6
LR	50.0	34.3	26.1	21.7	19.0	16.6	15.6	12.9
DTDR-C	88.6	67.7	62.6	60.3	63.4	54.6	48.8	49.7
DTDR-L	96.3	88.1	87.6	88.5	88.5	87.2	84.8	85.1
# Samples	1041	988	995	1032	988	564	263	105

Table 4: Top-1 tool retrieval accuracy (%) as a function of plan length in TinyAgent.

Performance improves with more data, plateauing around 10k samples. Gains are more pronounced for smaller LLM backbones. Notably, DTDR-L suffers a sharp drop below 1k samples due to overfitting, highlighting its sensitivity compared to the more robust DTDR-C.

How does accuracy scale with increasing Plan Length? We observe that in Table 4, DR remains relatively stable across plan lengths, as it conditions only on the most recent tool call. LR performs well on short plans but degrades as plan length increases. DTDR-C outperforms both baselines on shorter plans (length < 6) but declines on longer plans, likely due to limited data coverage for higher-order dependencies. DTDR-L consistently achieves the highest accuracy across lengths and generalizes better to longer plans, suggesting stronger extrapolation of dependency structure.

7 Conclusion

We introduce Dynamic Tool Dependency Retrieval (DTDR), a retrieval method that leverages both the tool dependencies and historical predictions via a retrieval module trained on demonstrations. DTDR has two variants: a supervised NN-based approach and an unsupervised clustering-based approach, both suitable for low-resource, on-device settings. Experiments across multiple datasets and LLM backbones show that DTDR significantly improves retrieval quality and downstream function selection over baselines.

8 Limitations

We wish to highlight some failure modes of DTDR:

- **Out-of-distribution generalization (DTDR-C).** DTDR-C performance degrades on out-of-distribution tasks, as both the clustering mechanism and higher-order Markov modeling require sufficient training data coverage to reliably estimate transition structure. Limited coverage reduces cluster quality and transition reliability.
- **Niche or non-standard tool vocabularies.** DTDR-C performance decreases on datasets with uncommon or highly domain-specific tool names and descriptions (e.g., HuggingFace-style function names). In such cases, pre-trained embeddings used for clustering may not adequately capture semantic similarity. A learnable model such as DTDR-L mitigates this issue by adapting its weights to better align with task-specific embedding representations.
- **Strong OOD tool dependencies.** All retrieval-based methods fail under strong out-of-distribution conditions, where tool names or dependency structures are entirely unseen during training. In such cases, an uncertainty-aware fallback mechanism could be applied: when the retriever's confidence is low, all tools are passed to the LLM instead of filtering. For DTDR-C, uncertainty can be estimated via cluster assignment likelihood; for DTDR-L, feature-space distances such as the Mahalanobis distance [1] can be used for OOD detection.

Additional failure modes can be found in Appendix E and F. Furthermore, this work assumes access to expert, ground-truth demonstrations. While demonstrations may be available as previously mentioned, having them be totally correct for all tasks may be a restrictive requirement. Therefore, future work can try to learn from failed, incorrect, or unlabeled samples. Other future directions include extending to multimodal tool-based tasks (e.g. robotics) and adapting to evolving tool sets.

References

Anthropic. 2025. Claude 4 system card: Opus and sonnet models. https://www.anthropic.com/research. Comprehensive technical and safety report for Claude 4 models.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. 2025. τ^2-bench: Evaluating conversational agents in a dual-control environment. *arXiv preprint arXiv:2506.07982*.

Norbert Braunschweiler, Rama Doddipatla, and Tudor-Catalin Zorila. 2025. Toolreagt: Tool retrieval for llm-based complex task solution via retrieval augmented generation. In *Proceedings of the 3rd Workshop on Towards Knowledgeable Foundation Models (KnowFM)*, pages 75–83.

Wenjie Chen, Wenbin Li, Di Yao, Xuying Meng, Chang Gong, and Jingping Bi. 2025. Gtool: Graph enhanced tool planning with large language model. *arXiv preprint arXiv:2508.12725*.

Charlie Cheng-Jie Ji, Mao Huanzhi, Yan Fanjia, G. Patil Shishir, Zhang Tianjun, Stoica Ion, and E. Gonzalez Joseph. 2024. Gorilla openfunctions v2.

Sijia Cui, Aiyao He, Shuai Xu, Hongming Zhang, Yanna Wang, Qingyang Zhang, Yajing Wang, and Bo Xu. 2025. Self-guided function calling in large language models via stepwise experience recall. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 10842–10854, Suzhou, China. Association for Computational Linguistics.

Hy Dang, Tianyi Liu, Zhuofeng Wu, Jingfeng Yang, Haoming Jiang, Tao Yang, Pei Chen, Zhengyang Wang, Helen Wang, Huasheng Li, and 1 others. 2025. Improving large language models function calling and interpretability via guided-structured templates. *arXiv preprint arXiv:2509.18076*.

Keyan Ding, Jing Yu, Junjie Huang, Yuchen Yang, Qiang Zhang, and Huajun Chen. 2025. Scitoolagent: a knowledge-graph-driven scientific agent for multitool integration. *Nature Computational Science*, pages 1–11.

Lutfi Eren Erdogan, Nicholas Lee, Siddharth Jha, Sehoon Kim, Ryan Tabrizi, Suhong Moon, Coleman Hooper, Gopala Anumanchipalli, Kurt Keutzer, and Amir Gholami. 2024. Tinyagent: Function calling at the edge. *arXiv preprint arXiv:2409.00608*.

Marco Federici, Davide Belli, Mart Van Baalen, Amir Jalalirad, Andrii Skliar, Bence Major, Markus Nagel, and Paul Whatmough. 2025. Efficient llm inference using dynamic input pruning and cache-aware masking. In *Eighth Conference on Machine Learning and Systems*.

Linfeng Gao, Yaoxiang Wang, Minlong Peng, Jialong Tang, Yuzhe Shang, Mingming Sun, and Jinsong Su. 2025. Tool graph retriever: Exploring dependency graph-based tool retrieval for large language models. *arXiv preprint arXiv:2508.05152*.

Google. 2025. Google Antigravity AI IDE.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, and 1 others. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*.

Qiqiang Lin, Muning Wen, Qiuying Peng, Guanyu Nie, Junwei Liao, Jun Wang, Xiaoyun Mo, Jiamu Zhou, Cheng Cheng, Yin Zhao, and 1 others. 2024. Hammer: Robust function-calling for on-device language models via function masking. *arXiv preprint arXiv:2410.04587*.

Xukun Liu, Zhiyuan Peng, Xiaoyuan Yi, Xing Xie, Lirong Xiang, Yuchen Liu, and Dongkuan Xu. 2024a. Toolnet: Connecting large language models with massive tools via tool graph. *arXiv preprint arXiv:2403.00839*.

Zhaoyang Liu, Zeqiang Lai, Zhangwei Gao, Erfei Cui, Ziheng Li, Xizhou Zhu, Lewei Lu, Qifeng Chen, Yu Qiao, Jifeng Dai, and 1 others. 2024b. Controlllm: Augment language models with tools by searching on graphs. In *European Conference on Computer Vision*, pages 89–105. Springer.

OpenAI. 2025. Gpt-5 system card. https://openai.com/research/index/publication/. Describes the architecture, safety measures, and capabilities of GPT-5.

Varatheepan Paramanayakam, Andreas Karatzas, Iraklis Anagnostopoulos, and Dimitrios Stamoulis. 2025. Less is more: Optimizing function calling for llm execution on edge devices. In *2025 Design, Automation & Test in Europe Conference (DATE)*, pages 1–7. IEEE.

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

Bhrij Patel, Ashish Jagmohan, and Aditya Vempaty. 2025. Learning api functionality from in-context demonstrations for tool-based agents. *Empirical Methods of Natural Language Processing*.

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

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. 2024. Gorilla: Large language model connected with massive apis. *Advances in Neural Information Processing Systems*, 37:126544–126565.

Shuofei Qiao, Honghao Gui, Chengfei Lv, Qianghuai Jia, Huajun Chen, and Ningyu Zhang. 2023. Making language models better tool learners with execution feedback. *arXiv preprint arXiv:2305.13068*.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, and 1 others. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789*.

Ella Rabinovich and Ateret Anaby-Tavor. 2025. On the robustness of agentic function calling. *arXiv preprint arXiv:2504.00914*.

Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics.

Stephen Robertson, Hugo Zaragoza, and 1 others. 2009. The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends® in Information Retrieval*, 3(4):333–389.

Vishnu Sarukkai, Zhiqiang Xie, and Kayvon Fatahalian. 2025. Self-generated in-context examples improve llm agents for sequential decision-making tasks. *arXiv preprint arXiv:2505.00234*.

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.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2023. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. *Advances in Neural Information Processing Systems*, 36:38154–38180.

Yongliang Shen, Kaitao Song, Xu Tan, Wenqi Zhang, Kan Ren, Siyu Yuan, Weiming Lu, Dongsheng Li, and Yueting Zhuang. 2024. Taskbench: Benchmarking large language models for task automation. *Advances in Neural Information Processing Systems*, 37:4540–4574.

Qingyu Song, Peiyu Liao, Wenqian Zhao, Yiwen Wang, Shoubo Hu, Hui-Ling Zhen, Ning Jiang, and Mingxuan Yuan. 2025. Harnessing on-device large language model: Empirical results and implications for ai pc. *arXiv preprint arXiv:2505.15030*.

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

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

Aohan Zeng, Xin Lv, Qinkai Zheng, Zhenyu Hou, Bin Chen, Chengxing Xie, Cunxiang Wang, Da Yin, Hao Zeng, Jiajie Zhang, and 1 others. 2025. Glm-4.5: Agentic, reasoning, and coding (arc) foundation models. *arXiv preprint arXiv:2508.06471*.

A Detailed Related Works

Training LLMs for Tool Use. Recent LLMs demonstrate impressive tool-usage capabilities, with cloud-based models such as GPT-5 (OpenAI, 2025), Claude 4 (Anthropic, 2025), and GLM-4.5 (Zeng et al., 2025) leading by a wide margin on function-call benchmarks like BFCL V4 (Patil et al., 2025) and τ^2-Bench (Barres et al., 2025). In contrast, smaller LLMs designed for edge deployment perform significantly worse on agentic tasks when evaluated out-of-the-box, creating a challenge for widespread adoption on mobile devices. To address this, recent work has explored fine-tuning LLMs on large function calling datasets, sourced from cloud providers (Cheng-Jie Ji et al., 2024), generated by cloud models (Qiao et al., 2023), or obtained through self-play in agentic environments (Schick et al., 2023). However, collecting tool-use data for fine-tuning is often prohibitively expensive, and these fine-tuned models frequently struggle to generalize to new or updated tools.

LLM Prompting with Tool Retrieval. An alternative line of research leverages the in-context learning capabilities of LLMs, introducing prompt-engineering strategies for tool learning. This can involve including tool descriptions (Shen et al., 2023, 2024) or example trajectories (Paranjape et al., 2023; Sarukkai et al., 2025) within the model prompt. To handle hundreds of apps and APIs, recent work uses retrieval-augmented generation to sub-select tools that are relevant to the task. ToolLLM (Qin et al., 2023) and ToolReAGt (Braunschweiler et al., 2025) identify the most suitable tools by comparing embeddings derived from the user query and tool documentation. Less-is-More (Paramanayakam et al., 2025) further prompts an LLM to generate documentation for the ideal tools to solve the query. ART (Paranjape et al., 2023) uses LLM prompting to retrieve example trajectories for most relevant tools.

Tool Dependency Retrieval. The execution of certain tools often depends on the outputs of others. ControlLLM (Liu et al., 2024b) and SciToolAgent (Ding et al., 2025) address specific scenarios where the tool dependency graph is known. In most real-world cases, however, the set of tools and their dependencies are not known in advance or may change at test time, for example when apps are installed or updated. In that case, tool dependencies could be learned from demonstration data provided by tool vendors (Paranjape et al., 2023), extracted from historical tool trajectories (Chen et al., 2025), generated procedurally (Erdogan et al., 2024; Shen et al., 2024), or synthesized using cloud LLMs (Gao et al., 2025; Patil et al., 2024; Qin et al., 2023). Tool Graph Retriever (Gao et al., 2025) and GTool (Chen et al., 2025) employ GNNs to encode the learned dependencies, using the resulting embeddings for similarity-based retrieval and for LLM prompting. In contrast, ToolNet (Liu et al., 2024a) parses the tool dependency graph to identify the set of eligible tools after the latest tool call in the plan. Rather than modeling a static tool graph from demonstration data, we propose a lightweight approach that dynamically retrieves specific tool dependencies which are relevant to the test query and the sequence of previously called tools.

B Dataset Statistics

In Table 5 we report important statistics for the datasets considered in our evaluation. While the average length of the plans is comparable across datasets, the number of tool dependencies varies significantly, which means that certain datasets benefit more than others in modeling and employing tool dependencies for function selection.

C Implementation Details

Please see Appendix I for pseudocode to implement these methods.

Tool Dependency Graph Construction in DTDR-C We compute the tool dependency graph based on the training data D_{train}. Each training datapoint contains a ground-truth trajectory for the given query. Let $f_l = (f_{i-o}, f_{i-l+1}, \cdots, f_i)$ be a history of function calls of length l. For example, a possible f_2 could be the tuple, ('get_email', 'reply_email'), where 'get_email' is the first function in the sequence and 'reply_email is the latest function. For each function sequence f_l used in the ground-truth trajectories of D_{train}, let $\mathcal{F}^{f_l}$ be the set of all unique functions that have appeared immediately after f_l. The tool dependency graph G_l is formatted according to whether it is probabilistic or not. When G_l is probabilistic, each f_l is a key in

Dataset	# Plans	# Tools	# Tool calls per plan	# Tool dependencies per plan
TinyAgent	39874	17	4.5±1.9	1.9±1.8
TaskBench DailyLife	3860	41	4.1±1.7	0.1±0.2
TaskBench HuggingFace	4959	24	3.2±1.5	1.1±1.5
TaskBench Multimedia	4310	41	3.6±1.7	1.5±1.7

Table 5: Function Calling Benchmark statistics. The number of tool dependencies refers to cases where the parameter of a function call is the output of a function call earlier in the plan. Notice that TaskBench DailyLife mainly contains plans with unrelated function calls.

Tool Dependency Graph G_6 **Key,** f_6	**Unweighted Tool, Dependency Graph Value,** F^{f_6}	**Weighted Tool Dependency Graph Value,** $P(f'\|f_6)$
(‘start’, ‘start’, ‘start’, ‘open_and_get_file_path’, ‘get_email_address’, ‘compose_new_email’)	[‘join’, ‘create_calendar_event’, ‘create_reminder’, ‘summarize_pdf’, ‘append_note_content’]	{‘join : 0.8, ‘create_calendar_event’ : 0.05, ‘create_reminder’ : 0.05, ‘summarize_pdf’: 0.05, ‘append_note_content’: 0.05 }

Table 6: **Example Tool Dependency Graph.** Given a function sequence of length 6, f_6 (left), we can compute based on demonstration data, D, the unweighted tool dependency graph (middle) and weighted tool dependency graph (right).

a nested dictionary. The value of each f_l is another dictionary representing a probability distribution where the inner keys are all the functions in $\mathcal{F}^{f_l}$. For each inner function key, $f' \in \mathcal{F}^{f_l}$, the value is, $P(f'|f_l)$: the percent of times f' appears given that f_l appears immediately before it. Thus, $P(f'|f_l)$ can be considered an l-ordered Markov Chain. If G_l is unweighted, then G_l is a flat dictionary where each f_l is a key and the corresponding value is $\mathcal{F}^{f_l}$. Table 6 gives and example of a tool dependency graph of order 6, or G_6. To pad function sequences with length less than order l, we use the string ‘start’ as a placeholder. In Table 6, the example f_6 only has three functions in the history, so we use use three ‘start’ strings in the beginning. In summary, we use a sliding window approach with window length l, and we construct G based on the next function right after the current window. For *DTDR-C*, we find the optimal number of clusters to be $1/10$ of the training set size.

Classifier optimization in DTDR-L Fine-tuning for TinyAgent and *DTDR-L* variants is performed using Adam with a learning rate of 10^{-3}, learning rate decay of 0.9, weight decay of 10^{-5}, and 10 epochs. For DTDR-L, the training of ϕ is per-step. For each training query $q \in Q_{train}$, we train our predictor ϕ using the sum of independent $|\mathcal{F}|$ binary cross-entropy loss terms. Formally, let $d_{0:t-1}$ be the possible ground-truth demonstration trajectory for q up until time t. Both $d_{0:t-1}$ and $f_{0:t-1}$ are a list of function names, so $d_{0:t-1}$ can be concatenated to q and passed into ϕ. Therefore, we can write

$$\phi^* = \arg\min_{\phi} - \sum_{q \in Q_{train}} \sum_{t=0}^{T} \sum_{f \in \mathcal{F}} (\mathbf{1}_{f \in \mathcal{F}^*_{q,t}})(\log \phi(q, d_{0:t-1})). \tag{2}$$

Other details and hyper-parameters For all Tool Retriever methods, we use the same sentence-embedding model *paraphrase-MiniLM-L6-v2* (Reimers and Gurevych, 2019) across all approaches to ensure comparability. The dimensionality of the embeddings $e_\theta = 384$. For *Less-is-More*, we ensure the same LLM is used both for Function Selection and for recommending the ideal tool descriptions. We consider Search Level 1 but not Level 2, as the latter needs interaction with a cloud model, which is prohibitive on-device. For *ToolNet*, we build the tool dependency graph from the training split and use it to retrieve and score relevant tools based on the latest tool call in the plan. For a fair comparison with *Tool Graph Retriever*, we also use the available example trajectories to

	Retrieval Method	Function Selection Acc. [%] (↑)	Retrieval Metrics MRR (↑)	F_1 (↑)
TinyAgent	Random Guess	5.9	0.26	0.12
	BM-25 (Robertson et al., 2009)	23.1	0.35	0.20
	QTS (Vanilla)	15.8	0.26	0.14
	QTS (Gao et al., 2025)	21.5	0.18	0.09
	QTS (Paramanayakam et al., 2025)	23.7	0.36	0.19
	DR (Liu et al., 2024a)	30.7	0.70	0.49
	Dynamic DR (DTDR-C) (Ours)	43.3	0.78	**0.56**
	LR (Erdogan et al., 2024)	25.6	0.53	0.39
	Dynamic LR (DTDR-L) (Ours)	**65.1**	**0.93**	0.55
Taskbench-HF	Random Guess	4.2	0.16	0.05
	BM-25 (Robertson et al., 2009)	8.3	0.18	0.05
	QTS (Vanilla)	7.9	0.21	0.08
	QTS (Gao et al., 2025)	14.6	0.37	0.27
	QTS (Paramanayakam et al., 2025)	12.9	0.30	0.18
	DR (Liu et al., 2024a)	17.1	0.46	0.33
	Dynamic DR (DTDR-C) (Ours)	27.5	0.64	0.55
	LR (Erdogan et al., 2024)	20.5	0.53	0.32
	Dynamic LR (DTDR-L) (Ours)	**27.8**	**0.75**	**0.63**
Taskbench-MM	Random Guess	2.4	0.11	0.03
	BM-25 (Robertson et al., 2009)	13.7	0.26	0.19
	QTS (Vanilla)	5.3	0.14	0.04
	QTS (Gao et al., 2025)	5.3	0.14	0.20
	QTS (Paramanayakam et al., 2025)	7.6	0.29	0.18
	DR (Liu et al., 2024a)	13.3	0.38	0.27
	Dynamic DR (DTDR-C) (Ours)	24.4	0.52	0.43
	LR (Erdogan et al., 2024)	21.0	0.49	0.28
	Dynamic LR (DTDR-L) (Ours)	**27.0**	**0.69**	**0.55**
Taskbench-DL	Random Guess	2.4	0.15	0.07
	BM-25 (Robertson et al., 2009)	13.5	0.31	0.15
	QTS (Vanilla)	7.5	0.16	0.07
	QTS (Gao et al., 2025)	13.3	0.17	0.06
	QTS (Paramanayakam et al., 2025)	18.9	0.40	0.26
	DR (Liu et al., 2024a)	14.7	0.36	0.18
	Dynamic DR (DTDR-C) (Ours)	28.2	0.54	0.29
	LR (Erdogan et al., 2024)	23.6	0.55	0.44
	Dynamic LR (DTDR-L) (Ours)	**58.9**	**0.85**	**0.64**

Table 7: Retrieval performance for methods from different categories (QTS = Query-Tool Similarity, LR = Learned Retriever, DR = Dependency Retriever). QTS and LR categories are query-conditioned, while DR is history-conditioned. Our Dynamic methods are conditioned on both query and tool history. Function Selection Accuracy is reported on Qwen 3 0.6B using hard masking as ICL method. Best results per dataset and metrics are in bold.

build the tool dependency graph, instead of relying on a separate predictor. For the *TinyAgent* retriever, we fine-tune a linear layer on top of the sentence embedding model to score all tools given the test query. For all Tool Retrievers other than *DTDR-C* (which does not require it) we sweep the function selection threshold with values of $0.2, 0.23, 0.5$ and set it based on function selection accuracy. For ICL experiments we use greedy decoding with a maximum generation length matching the longest tool name in the dataset. For the *Raw Demonstrations* ICL method, we provide 5 random demonstrations

	Function Selection Accuracy [%] (↑)				
	Method	TinyAgent	TaskBench DailyLife	TaskBench HuggingFace	TaskBench Multimedia
Qwen3 0.6B	No ICL	23.2	12.8	5.5	7.3
	Dependency Retrieval (Liu et al., 2024a)	22.0	11.5	6.7	9.5
	Learned Retrieval (Erdogan et al., 2024)	19.8	19.6	10.0	12.1
	DTDR-C (Ours)	35.3	23.7	**20.8**	**20.2**
	DTDR-C: Raw Dems Prompting (Ours)	17.2	10.5	4.3	5.3
	DTDR-L (Ours)	**46.1**	**45.9**	18.9	17.6
Qwen3 1.7B	No ICL	48.1	66.7	31.2	38.1
	Dependency Retrieval (Liu et al., 2024a)	51.6	60.0	41.6	45.1
	Learned Retrieval (Erdogan et al., 2024)	50.5	56.6	42.8	42.1
	DTDR-C (Ours)	64.1	60.2	**52.8**	**47.8**
	DTDR-C: Raw Dems Prompting (Ours)	46.3	69.0	43.1	46.9
	DTDR-L (Ours)	**78.4**	**83.1**	49.0	45.5
Qwen3 4B	No ICL	59.6	83.5	45.8	54.3
	Dependency Retrieval (Liu et al., 2024a)	62.7	64.7	52.8	51.8
	Learned Retrieval (Erdogan et al., 2024)	52.2	60.4	56.9	58.2
	DTDR-C (Ours)	68.1	64.7	56.0	52.4
	DTDR-C: Raw Dems Prompting (Ours)	60.2	87.6	49.8	57.3
	DTDR-L (Ours)	**80.7**	**89.0**	**60.5**	**64.1**
Qwen3 8B	No ICL	71.3	93.9	70.5	78.5
	Dependency Retrieval (Liu et al., 2024a)	76.7	75.3	67.3	69.1
	Learned Retrieval (Erdogan et al., 2024)	43.7	52.2	52.0	49.6
	DTDR-C (Ours)	80.4	75.3	67.3	69.1
	DTDR-C: Raw Dems Prompting (Ours)	67.8	**94.5**	**72.5**	**80.4**
	DTDR-L (Ours)	**89.0**	91.0	63.8	67.6
Qwen3 14B	No ICL	78.0	96.2	74.4	81.7
	Dependency Retrieval (Liu et al., 2024a)	79.5	84.1	69.2	73.8
	Learned Retrieval (Erdogan et al., 2024)	51.1	56.5	53.5	50.8
	DTDR-C (Ours)	84.8	84.1	71.5	74.0
	DTDR-C: Raw Dems Prompting (Ours)	77.9	**96.6**	**76.7**	**82.3**
	DTDR-L (Ours)	**92.5**	93.7	71.2	73.4
Gorilla V2	No ICL	39.5	PLE	PLE	PLE
	Dependency Retrieval (Liu et al., 2024a)	32.2	PLE	PLE	PLE
	Learned Retrieval (Erdogan et al., 2024)	37.9	PLE	PLE	PLE
	DTDR-C (Ours)	43.4	PLE	PLE	PLE
	DTDR-C: Raw Dems Prompting (Ours)	PLE	PLE	PLE	PLE
	DTDR-L (Ours)	**58.7**	PLE	PLE	PLE
GPT-4o	No ICL	84.7	96.6	74.8	81.6
	Dependency Retrieval (Liu et al., 2024a)	85.9	82.9	70.0	72.8
	Learned Retrieval (Erdogan et al., 2024)	53.2	48.1	55.2	51.1
	DTDR-C (Ours)	85.9	82.9	71.9	74.8
	DTDR-C: Raw Dems Prompting (Ours)	86.8	**96.8**	**77.1**	**82.3**
	DTDR-L (Ours)	**94.2**	91.5	71.8	73.7

Table 8: Comparison of selected methods in terms of function selection accuracy over different datasets and models. Best results per dataset and model are bolded. Gorilla V2 has 4096 maximum context length, which easily results in PLE (Prompt Length Exceeded).

as examples. When defining train and test splits for each dataset, we ensure that the number of occurrences in train and test data for each tool follows then 70/30 ratio. We use FP16 precision for Qwen models and Gorilla-V2, while GPT-4o is prompted through cloud APIs. All experiments are conducted on NVIDIA A100 GPUs.

D Metric Details

For downstream performance, we measure *Function Selection Accuracy* as described in (Erdogan et al., 2024): for each step, the selected function is considered correct only it belongs to the set of acceptable tools according to the Directed Acyclic Graph (DAG) of the ground-truth plan. For *Success Rate* we evaluate both function selection *and* predicted parameters for *all* steps in the plan. The success rate is 1 only if the DAG of the plan is isomorphic to the DAG of the ground-truth plan (Erdogan et al., 2024). The same LLM backbone is prompted to predict the function calling parameters given the query and selected function. Example of prompts for function selection and parameter filling are provided in Appendix G. Tool Retrieval methods are evaluated with ranking and retrieval metrics. *Mean Reciprocal Rank (MRR)* measures the relative ranking of tools. F_1 score is the harmonic mean of Precision and Recall and is computed considering the top-k highest-scoring tools as retrieved, with k being the number of acceptable tools according to the DAG of the ground-truth plan.

E Additional Results

Retrieval Accuracy We report the full values for retrieval and function selection accuracy with Qwen 3 0.6B with unweighted Hard Masking. DTDR consistently outperforms in both retireval and function selection across datasets. We note that for Taskbench-Multimedia that LR outperforms in function selection due to overfitting to repeatedly predict "end-of-plan" which is always the last function in each task.

Function Selection Accuracy Across Models We report here the full results on function selection accuracy discussed in the Results and Discussion Section. Additional LLM backbones include Qwen 3 14B and Gorilla-v2. At similar footprints, Gorilla V2 under-performs Qwen 3 8B, likely due to differences between our test data and the coding API data used for Gorilla's fine-tuning. This difference in performance reinforces our claim that adaptive ICL strategies are preferable to fine-tuned generalist models, as ICL can seamlessly adapt to unseen tools at test time.

F Weighted vs Unweighted Masking

To see the impact between weighted and unweighted masking, we compute the percentage of correct predictions when the $DTDR - C$-based ω did not retrieve any $f \in F_t^*$ with a probability above threshold 0.5. We see for Tinyagent across all models and number of cluster settings, Hard Masking unweighted achieves the maximum of around 33.6% and outperforms hard mask weighted by up to 10%. This difference is also present when comparing Soft Masking with its weighted variant, and this difference shows that when the demonstration data does not reflect the task well enough, the weights provided in the prompt mislead the agent. However, as the number of clusters increase, this difference between in values for weighted and unweighted decreases. Similar observations are present for probability thresholds of 0.4 and 0.3.

G Prompt Examples

We provide example prompts showing guidelines and ICL techniques used for function selection and parameter filling. Grayed out comments are only used as reference, and are not included in the actual prompt.

Full Prompt G.1: Function Selection

```
# System Guidelines
You are an expert AI Agent specialized in
    creating plans comprised of function
    calls to
satisfy user queries.
You are given a user query and a set of
    functions that should be used to
    solve the query.
You are also given the executed plan so
    far to solve the query.
You task is to only select one of the
    given functions as the immediate next
     step to be added to the plan.
You MUST only output one of the functions
    among the function list.
You MUST NOT include any other text in the
     response.
You are given below an example query along
     with the executed plan paired with
    the expected answer to solve the query
    .
You can use this example to learn to
    predict the correct function to add
    to the plan for the user query.

# Example Functions
Set of available functions and their
    descriptions:
```

```
[{'function': 'create_calendar_event', '
    description': 'Create a calendar
    event'},
...,
{'function': 'summarize_pdf', 'description
    ': 'Summarizes the content of a PDF
    file and returns the summary'}]
You will choose one of the function names
    above.

# Example Task
User query: 'Open the file "ProjectPlan.
    docx", create a reminder to review it
     by next Monday, and find directions
    to the nearest cafe in Apple Maps.'.
Executed plan so far:
out_1 = open_and_get_file_path(
    name_or_path='ProjectPlan.docx')

# Example Answer
create_reminder

# Test Functions
Set of available functions and their
    descriptions:
[{'function': 'create_calendar_event', '
    description': 'Create a calendar
    event'},
...,
{'function': 'summarize_pdf', 'description
    ': 'Summarizes the content of a PDF
    file and returns the summary'}]
You will choose one of the function names
    above.

# Test Task
Now choose the next function call for this
     task:
User query: 'Reply to the currently
    selected email in Mail with the match
     details attached and create a new
    note titled "Festival Notes" to
    summarize the discussions.'.
Executed plan so far: None
```

Full Prompt G.2: Parameter Filling

```
# System Guidelines
You are an expert AI Agent specialized in
    calling functions to satisfy user
    queries.
You are given a user query you are trying
    to complete.
You are also given the function to be used
     and a description of its behavior
    and input parameter.
Based on this, you should identify which
    variables (words in the query) to
    assign to each function parameter in
    order to solve the query.
You should only return the parameter names
     and associated variables.
You MUST put the matched parameter names
    and variables in the format of
    arg_name1=variable1, arg_name2=
    variable2, ...
The parameter names MUST be among the ones
     listed in the Input.
For optional parameter, defined with
    is_optional=True, you can choose to
    either assign a variable, e.g.: '
    arg_name1=variable1,' or instead
    leave the parameter unused.
You SHOULD NOT include any other text in
    the response.
You are given below one example query for
    the same function, paired with the
    expected answer to solve the query.
You can use this as an example to predict
    the correct parameter and variables
    matching for the query.

# Example Task
Input:
User query: 'Open the location "The
    Peninsula, New York" in Apple Maps
    and append the summary of the "
    TravelGuide.pdf" to the note titled "
    Vacation Plans" in the "Travel" folder
    .'.
Executed function calls so far with their
    respective output variable:
out_1 = maps_open_location(query='The
    Peninsula, New York')
out_2 = open_and_get_file_path(
    name_or_path='TravelGuide.pdf')
out_3 = summarize_pdf(pdf_path='out_2')
You will be using the function '
    append_note_content' to solve this
    query. What this function does is '
    Appends content to an existing note'.
This function accepts the following
    parameters names, each listed with
    its expected variable type, whether
    it is optional, and a short
    description.
[name: 'name', type: 'str', is_optional: '
    False', description: ''], [name: '
    append_content', type: 'str',
    is_optional: 'False', description:
    ''], [name: 'folder', type: 'str',
    is_optional: 'True', description: '']
You will pair each of the parameter names
    above with one variable (or possibly
    an empty string, if optional).
Output:

# Example Answer
name=Vacation Plans, append_content=out_3,
     folder=Travel

# Test Task
Test query:
Input:
User query: 'Summarize the "Project
    Proposal.pdf" report, create a
    reminder to review it by next Tuesday
     at 2:00 PM, append key points to the
     note "Project Ideas" in the "Ideas"
    folder.'.
Executed function calls so far with their
    respective output variable:
out_1 = open_and_get_file_path(
    name_or_path='Project Proposal.pdf')
```

```
out_2 = summarize_pdf(pdf_path='out_1')
out_3 = create_reminder(name='Review
    Project Proposal', due_date
    ='2024-02-27 14:00:00', notes='',
    list_name='', priority=0, all_day=
    False)
You will be using the function '
    append_note_content' to solve this
    query. What this function does is '
    Appends content to an existing note'.
This function accepts the following
    parameters names, each listed with
    its expected variable type, whether
    it is optional, and a short
    description.
[name: 'name', type: 'str', is_optional: '
    False', description: ''], [name: '
    append_content', type: 'str',
    is_optional: 'False', description:
    ''], [name: 'folder', type: 'str',
    is_optional: 'True', description: '']
You will pair each of the parameter names
    above with one variable (or possibly
    an empty string, if optional).
Output:
```

ICL Prompt G.1: No ICL

```
# Test Functions
Set of available functions and their
    descriptions:
[{'function': 'create_calendar_event', '
    description': 'Create a calendar
    event'}, {'function': '
    get_phone_number', 'description': '
    Search for a contact by name. Returns
     the phone number of the contact.'},
    {'function': 'get_email_address', '
    description': 'Search for a contact by
     name. Returns the email address of
    the contact'}, {'function': '
    compose_new_email', 'description': '
    Composes a new email and returns the
    status of the email composition'}, {'
    function': 'reply_to_email', '
    description': 'Replies to the
    currently selected email in Mail with
     the given content'}, {'function': '
    forward_email', 'description': '
    Forwards the currently selected email
     in Mail with the given content'}, {'
    function': 'maps_open_location', '
    description': 'Opens the specified
    location in Apple Maps'}, {'function':
     'maps_show_directions', 'description
    ': 'Show directions from a start
    location to an end location in Apple
    Maps'}, {'function': 'create_note', '
    description': 'Creates a new note
    with the given content'}, {'function':
     'open_note', 'description': 'Opens
    an existing note by its name'}, {'
    function': 'append_note_content', '
    description': 'Appends content to an
    existing note'}, {'function': '
    create_reminder', 'description': '
    Creates a new reminder and returns
    the status of the reminder creation'},
     {'function': 'send_sms', '
    description': 'Send an SMS to a list
    of phone numbers'}, {'function': '
    open_and_get_file_path', 'description
    ': 'Opens the file and returns its
    path'}, {'function': '
    get_zoom_meeting_link', 'description':
     'Creates a Zoom meeting and returns
    the join URL'}, {'function': '
    summarize_pdf', 'description': '
    Summarizes the content of a PDF file
    and returns the summary'}, {'function
    ': 'join', 'description': 'Collects
    and combines results from prior
    actions. It should always be the last
     action in the plan'}]
You will choose one of the function names
    above.

# Test Task
User query: 'Create a new note titled "
    Recipe Ideas" in the "Cooking" folder
     with a list of ingredients for a
    lasagna recipe. Then, append the
    steps for preparation to the same
    note. Finally, send an SMS to
    yourself with a reminder to buy the
    ingredients for the lasagna.'.
Executed plan so far:
out_1 = create_note(name='Recipe Ideas',
    content='Lasagna Ingredients: -
    Ground beef - Lasagna noodles -
    Ricotta cheese - Mozzarella cheese -
    Parmesan cheese - Tomato sauce -
    Garlic - Onion - Olive oil - Salt -
    Pepper - Italian seasoning', folder='
    Cooking')
```

ICL Prompt G.2: Soft Mask

```
Set of available functions and their
    descriptions:
[{'function': 'create_calendar_event', '
    description': 'Create a calendar
    event'}, {'function': '
    get_phone_number', 'description': '
    Search for a contact by name. Returns
     the phone number of the contact.'},
    {'function': 'get_email_address', '
    description': 'Search for a contact by
     name. Returns the email address of
    the contact'}, {'function': '
    compose_new_email', 'description': '
    Composes a new email and returns the
    status of the email composition'}, {'
    function': 'reply_to_email', '
    description': 'Replies to the
    currently selected email in Mail with
     the given content'}, {'function': '
    forward_email', 'description': '
    Forwards the currently selected email
     in Mail with the given content'}, {'
    function': 'maps_open_location', '
    description': 'Opens the specified
```

```
    location in Apple Maps'}, {'function':
     'maps_show_directions', 'description
    ': 'Show directions from a start
    location to an end location in Apple
    Maps'}, {'function': 'create_note', '
    description': 'Creates a new note
    with the given content'}, {'function':
     'open_note', 'description': 'Opens
    an existing note by its name'}, {'
    function': 'append_note_content', '
    description': 'Appends content to an
    existing note'}, {'function': '
    create_reminder', 'description': '
    Creates a new reminder and returns
    the status of the reminder creation'},
     {'function': 'send_sms', '
    description': 'Send an SMS to a list
    of phone numbers'}, {'function': '
    open_and_get_file_path', 'description
    ': 'Opens the file and returns its
    path'}, {'function': '
    get_zoom_meeting_link', 'description':
     'Creates a Zoom meeting and returns
    the join URL'}, {'function': '
    summarize_pdf', 'description': '
    Summarizes the content of a PDF file
    and returns the summary'}, {'function
    ': 'join', 'description': 'Collects
    and combines results from prior
    actions. It should always be the last
     action in the plan'}]
You will choose one of the function names
    above.

# Soft Mask guidelines
To help guide your decision, here is a set
     of functions extracted from ground
    truth sequences of training queries.
The functions in this list are functions
    that have come right after your
    latest function call history of "('
    start', 'create_note')" in the
    training queries. The list is below:
['append_note_content', 'join']
Please use this to guide your decision-
    making on function selection.

# Test Task
User query: 'Create a new note titled "
    Recipe Ideas" in the "Cooking" folder
     with a list of ingredients for a
    lasagna recipe. Then, append the
    steps for preparation to the same
    note. Finally, send an SMS to
    yourself with a reminder to buy the
    ingredients for the lasagna.'.
Executed plan so far:
out_1 = create_note(name='Recipe Ideas',
    content='Lasagna Ingredients: -
    Ground beef - Lasagna noodles -
    Ricotta cheese - Mozzarella cheese -
    Parmesan cheese - Tomato sauce -
    Garlic - Onion - Olive oil - Salt -
    Pepper - Italian seasoning', folder='
    Cooking')
```

ICL Prompt G.3: Hard Mask

```
# Test Functions after Hard Mask
Set of available functions and their
    descriptions:
[{'function': 'append_note_content', '
    description': 'Appends content to an
    existing note'}, {'function': 'join',
     'description': 'Collects and
    combines results from prior actions.
    It should always be the last action
    in the plan'}]
You will choose one of the function names
    above.

# Test Task
User query: 'Create a new note titled "
    Recipe Ideas" in the "Cooking" folder
     with a list of ingredients for a
    lasagna recipe. Then, append the
    steps for preparation to the same
    note. Finally, send an SMS to
    yourself with a reminder to buy the
    ingredients for the lasagna.'.
Executed plan so far:
out_1 = create_note(name='Recipe Ideas',
    content='Lasagna Ingredients: -
    Ground beef - Lasagna noodles -
    Ricotta cheese - Mozzarella cheese -
    Parmesan cheese - Tomato sauce -
    Garlic - Onion - Olive oil - Salt -
    Pepper - Italian seasoning', folder='
    Cooking')
```

ICL Prompt G.4: Hard (Weighted)

```
# Test Functions after Hard Mask + Weights
Set of available functions, their
    descriptions, and the percentage of
    times they appear in the ground truth
     sequences of training queries right
    after this current function call
    history:
[{'function': 'append_note_content', '
    description': 'Appends content to an
    existing note', 'probability': 0.714},
     {'function': 'join', 'description':
    'Collects and combines results from
    prior actions. It should always be
    the last action in the plan', '
    probability': 0.286}]
You will choose one of the function names
    above.

# Test Task
User query: 'Create a new note titled "
    Recipe Ideas" in the "Cooking" folder
     with a list of ingredients for a
    lasagna recipe. Then, append the
    steps for preparation to the same
    note. Finally, send an SMS to
    yourself with a reminder to buy the
    ingredients for the lasagna.'.
Executed plan so far:
out_1 = create_note(name='Recipe Ideas',
    content='Lasagna Ingredients: -
    Ground beef - Lasagna noodles -
```

```
    Ricotta cheese - Mozzarella cheese -
    Parmesan cheese - Tomato sauce -
    Garlic - Onion - Olive oil - Salt -
    Pepper - Italian seasoning', folder='
    Cooking')
```

H Use of Large Language Models

Microsoft Co-Pilot was used for searching for related works and providing edit suggestions to the manuscript.

I Pseudocode

While we are currently unable to release the full implementation, we aim to support reproducibility by providing detailed pseudocode that outlines the experimental pipeline, as well as the methods and baselines employed.

I.1 Overall Experimental Pipeline

Algorithm 1 detail the overall pipeline on how we select multiple function names and subsequently Algorithm 2 details how we fill in the parameters for end-to-end function calling.

Algorithm 1 Get Function Selection Trajectory

Output: func_trajectory: list of selected function names for task
Input: query2demonstration: dictionary containing query key with corresponding value being a ground-truth trajectory demonstration; q: test query; T: ground-truth trajectory of q; M: agent model
Initialize empty list func_history $\leftarrow$ []
Initialize empty string new_func $\leftarrow$ ""
Initialize iter $\leftarrow 0$
while new_func $\neq$ "end" or iter $<$ MAX_ITERS **do**
    func_history $\leftarrow T[: iter]$
    Get retrieved functions ω using either retrieval method; see Sections I.2 and I.3
    Construct prompt p using Function Selection Prompt and ICL Method (See Appendix G for more details)
    Get next_func $\leftarrow M(p)$
    iter $\leftarrow$ iter $+ 1$
    Append next_func to func_trajectory
end while
Return: func_trajectory

Algorithm 2 Parameter Filling

Output: func_param_trajectory: list of dictionaries
Input: q: test query; func_trajectory: list of selected function names so far; func2signatures: dictionary of containing parameter names, types, and constraints for each functin name; M: agent model
Initialize func_param_history $\leftarrow$ []
Initialize next_params $\leftarrow \{\}$
Initialize iter $\leftarrow 0$
while iter $<$ len(func_trajectory) **do**
    func_history $\leftarrow T[:$ iter$]$
    Set target_func $\leftarrow$ func_trajectory[iter]
    Set target_signature $\leftarrow$ func2signatures[target_func]
    Get parameter filling prompt p (See Appendix G for more details)
    Get next_params = $M(p)$
    Append to next_params to func_param_trajectory
    iter $\leftarrow$ iter $+ 1$
end while
Return: func_param_trajectory

I.2 DTDR

DTDR-C. Algorithm 3 describes the implementation for building an N-order Markov Chain from demonstrations used for DTDR-C, and Algorithm 4 explains DTDR-C using pre-trained clusters from

k-Means. **Static Dependency Retrieval** is implemented by setting the number of clusters $k = 1$ and using order $N = 1$.

Algorithm 3 Build N-order Markov Chain, see Table 6 for an example

Output: G: nested dictionary of type Dict[Tuple[string] T, Dict[string s, float f]], where T is the previous N function names; s is a function name; float f is a value between 0 and 1.
Input: D ground-truth demonstration data; N: order of the Markov Chain.
Initialize $G \leftarrow \{\}$
for trajectory in D **do**
  for $0 <=$ index $< L$, length of trajectory **do**
    if iter $< N$ **then**
      Set left-padded $T =$ tuple(['start'] * ($N-$iter) + trajectory[:index])
    else
      Set $T =$ tuple(trajectory[index-(N-1):index])
    end if
    if $T \notin G$ **then**
      $G[T] \leftarrow \{\}$
    end if
    Set next function $s =$ trajectory[index+1]
    if $s \notin G[T]$ **then**
      $G[T][s] \leftarrow 0$
    end if
    Add occurence to G, $G[T][s] \leftarrow G[T][s] + 1$
  end for
end for
for $T \in G$ **do**
  Normalize the values in $G[T]$ to sum to 1
end for
Return: G

DTDR-L. Algorithm 5 describes the implementation for training the 1-linear-layer function classifier and Aglorithm 6 explains the implementation for DTDR-L. **Static Learned Retrieval** is trained using only the query as input. Therefore, it is done only once at the beginning of task in Algorithm 1.

I.3 Query-Tool Similarity Baselines.

Algorithm 7 explains the implementation for the Query-Tool Similarity (QTS) retrieval methods. The following methods include vanilla QTS, Less-is-More Level 1, and Tool Graph Retriever.

Algorithm 4 DTDR-Clustering

Output: ω: dictionary containing function name key and float value between 0 and 1. The sum of all float values should be 1.
Input: query2demonstration: dictionary containing query key with corresponding value being a ground-truth trajectory demonstration; q: test query; func_history: list of selected function names so far; N: order of the Markov Chain; E: embedding model; embedding2query: dictionary where keys are embeddings of training queries outputted by E and values are the original input queries; C: pre-trained K clusters of embeddings
Set test_embedding = $E(q)$
Find the closest cluster index $0 < k < K$ to test_embedding
Initialize list of demonstrations $D_k \leftarrow []$
for embedding $\in C_{\text{closest}}$ **do**
    Get string query $\leftarrow$ embedding2query[embedding]
    Append query2demonstration[query] to D_c
end for
Get N-order Markov Chain G using Algorithm 3
Set $\omega \leftarrow G$[tuple(func_history)]
Return: ω

Algorithm 5 Train linear layer for function predictino

Output: ϕ: trained 1-linear-layer classifier
Input: query2demonstration: dictionary containing query key with corresponding value being a ground-truth trajectory demonstration; E: embedding model; func_history: list of selected function names so far
for query, demonstration_trajectoy $\in$ query2demonstration **do**
    for $0 <=$ iter $< L$, length of demonstration_trajectory **do**
        Concatenate $x \leftarrow$ query + demonstration_trajectory[0:iter-1]
        Get embedding $e \leftarrow E(x)$
        Get softmax_output $\leftarrow Z(e)$
        Train ϕ using Equation 2
    end for
end for
Return: ϕ

Algorithm 6 DTDR-Linear

Output: ω: dictionary containing function name key and float value between 0 and 1. The sum of all float values should be 1.
Input: E: embedding model; q: test query; func_history: list of selected function names so far; α: float threshold; ϕ: trained function name classifier using Algorithm 5
Concatenate $x \leftarrow q+$ func_history
Get test_embedding $\leftarrow E(x)$
Get softmax_output $\leftarrow \phi$(test_embedding)
Get func2softmax, dictionary mapping function name to value from softmax_output
Initialize $\omega \leftarrow \{\}$
for func_name, softmax in func2softmax **do**
    if softmax $> \alpha$ **then**
        ω[func_name] $\leftarrow$ softmax
    end if
end for
Normalize values in ω to sum up to 1
Return: ω

Algorithm 7 Query Tool Similarity Baselines, **Note:** Unlike DTDR and static DR, the QTS baselines are only done ONCE at the beginning of the task. The retrieved set of functions is constant throughout the function selection steps

Output: ω: dictionary containing function name key and float value between 0 and 1. The sum of all float values should be 1.
Input: func_descriptions: dictionary containing function name key and the corresponding documentation; E: embedding model; q: test query; qts_method: string specifying which QTS baseline to use; M: agent model to generate what the agent thinks are the relevant tools for q if qts_method == "less_is_more"; G_T : matrix representing a tool dependency graph. We obtain G_T via Algorithm 3 by setting $N = 1$ and $K = 1$; α: float threshold
if qts_method == "less_is_more" **then**
    Set Less is More Thought Prompt p
    Set string_to_embed $\leftarrow M(q, p)$
else
    string_to_embed $\leftarrow q$
end if
test_embedding $\leftarrow E$(string_to_embed)
Set func_description_embedding $\leftarrow \{\}$
for func_name, description $\in$ func_description **do**
    func_description_embeddings[func_name] $\leftarrow E$(description)
end for
if qts_method == "tool_graph_retrieval" **then**
    Update func_description_embeddings with G_T using convolution operation
end if
Initialize query_tool_distances $\leftarrow \{\}$
for func_name, func_embedding $\in$ func_description_embeddings **do**
    query_tool_distances[func_name] $\leftarrow$ cosine_similarity(test_embedding, func_embedding)
end for
Initialize $\omega \leftarrow \{\}$
for func_name, similarity in query_tool_distances **do**
    if similarity $> \alpha$ **then**
        ω[func_name] $\leftarrow$ similarity
    end if
end for
Normalize values in ω to sum up to 1
Return: ω