Dynamic Tool Dependency Retrieval for Efficient Function Calling

Anonymous authors
Paper under double-blind review

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 execution context. 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 dynamic tool retrieval improves function calling success rates between 23% and 104% compared to state-of-the-art static retrievers.

1 Introduction

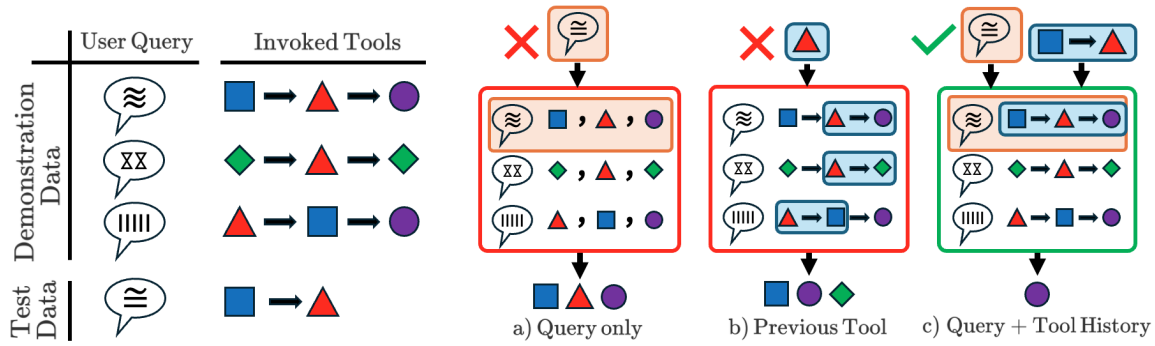

Figure 1: **Dynamic Tool Dependency Retrieval.** Given demonstration data for a set of tools, previous work retrieve tools based on either a) the natural language query (highlighted in orange) or b) the latest executed tool call in the plan (the red triangle, highlighted in teal). We instead propose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

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) *efficiency* under strict memory and latency budgets, and (ii) *effectiveness* across large and heterogeneous tool sets.

Therefore, prior work have 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 contextual relevance of selected tools. Others leverage static tool-dependency graphs built from

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

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)	✓	✓	✓	✓	✓

demonstration trajectories (Liu et al., 2024a), but these approaches risk retrieving tools irrelevant to the query or being biased toward repeated calls. For retrieval to provide the most informed context, it 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.
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 calling 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 Prompt Encoding 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 purposes function calling models (Cheng-Jie Ji et al., 2024). These model often fail in real-world scenarios due to poor tool selection, misinterpretation of user intent, or under realistic data perturbations (Dang et al., 2025; Rabinovich & 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 work retrieve 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 sequence of previously invoked tools. A more extensive discussion of related works can be found in Appendix A.

In Table 1 we summarize related works that most closely align with our method. We compare against 5 different categories. **1) Tool Dependency (Deps.) Aware:** retrievers that utilize tool dependencies are better informed to solve multi-step tasks (Gao et al., 2025); **2) Tool Description (Desc.) 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 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.

3 Problem Formulation

Retrieval-Augmented Function Selection. Consider the tool-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 $q \in Q$, such as *"Reply to my latest email from Willem."*. Let $f_{q,0:t-1} = [f_0, f_1, ..., f_{t-1}]$ be the list of previous functions predicted for q by π up until timestep t. Typically, q, $f_{q,0:t-1}$ and $\mathcal{F}$ are encoded in the LM prompt, p, which is input to the agent to sample a function $f_t \sim \pi(\cdot|p(q, f_{q,0:t-1}, \mathcal{F}))$. A sampled function f_t is correct if $f_t \in \mathcal{F}^*_{q,t}$, where $\mathcal{F}^*_{q,t}$ is the set of correct functions for the given query q at timestep t. To raise the probability of $f_t \in \mathcal{F}^*_{q,t}$, 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. Once again for notation simplicity, we drop q for $f_{0:t} := f_{q,0:t}$, $\mathcal{F}^*_t := \mathcal{F}^*_{q,t}$. Let T be the maximum trajectory length for any $q \in Q$, let Ω be the set of all possible ω retriever functions, and let $\mathbf{1}$ be a scalar indicator function. Thus, our function selection problem is a prompt optimization objective:

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

This optimization problem provides a mathematical grounding between the success of the function calling agent π and how well the retrieved distribution $\mathcal{F}_t$ 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: $\omega^*(\cdot) = \mathcal{F}_t \subseteq \mathcal{F}^*_t, |\mathcal{F}_t| > 0$. 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.

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. Therefore, let f_{desc} be the description of f, and letting $\mathcal{F}_{desc} = \{f_{desc} \mid \forall f \in \mathcal{F}\}$, the retriever module ω becomes a function of q and $\mathcal{F}_{desc}$: $\omega\,(q, \mathcal{F}_{desc})$. Furthermore, the retrieved set $\mathcal{F}_t$ is used throughout the entire trajectory, constant throughout time steps $t \in [0, T]$. Relying solely on static tool descriptions 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, *"get_email_address"* must be called consecutively multiple times. Demonstrations that contain function trajectories such as these will cause a high value of $P(\text{"get\_email\_address"}|\text{"get\_email\_address"})$, 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 $P(\text{"get\_email\_address"}|\text{"get\_email\_address"})$ value suitable in tasks with multiple recipients can be misleading in tasks with a single recipient. Without the task-specific information, the context of previous functions may be a misleading prior for π.

In this work, we aim for a *lightweight retrieval mechanism to obtain tool-dependency priors that capture both task-specific and context-specific information* for on-device function calling.

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, $\omega(q, f_{0:t-1})$.

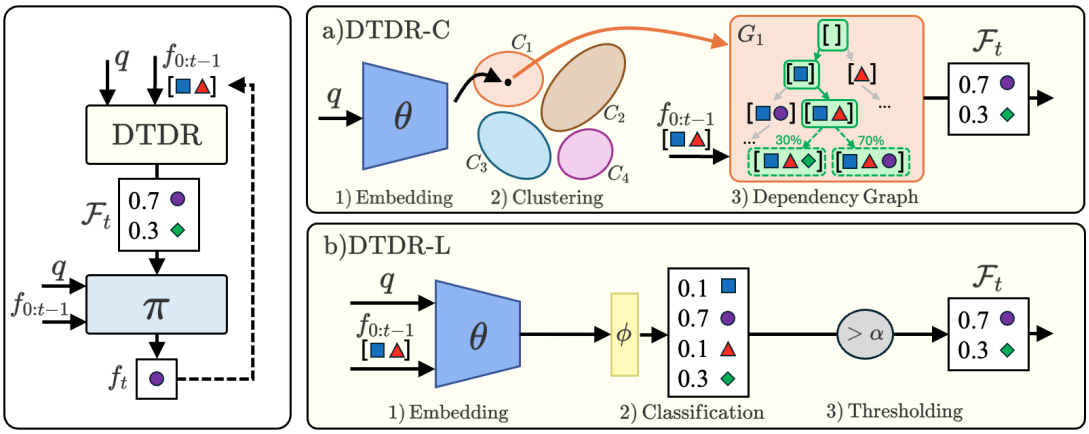

Figure 2: **System diagram for DTDR**. On the left, the user query and tool history are input to DTDR to retrieve the most likely next tools. The LLM π selects the next tool among this set. On the right, we show the two alternative instantiations for the retriever: a) DTDR-C, based on a clustering step 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.

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 $\mathcal{F}_t := \omega(q, f_{0:t-1})$, assigning a probability to each. We then encode the retrieved dependencies into the prompt $p(q, f_{0:t-1}, \mathcal{F}_t)$ 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 $f_t \sim \pi(\cdot|p(q, f_{q,0:t-1}, \mathcal{F}))$. For end-to-end evaluation, this continues in an iterative approach starting from the empty function call plan $f_\emptyset = []$, and ending after predicting the "end-of-plan" function $f_t =$ *"end"*.

4.1 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 of the two variants might be preferred for a real on-device scenario. Both proposed 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 θ and fit a K-Means clustering model C on the embeddings. For each cluster k with $0 \leq k < K$, we consider the set of demonstrations assigned to that cluster $D_k = \{d \mid q_d \in Q_D,\ C(\theta(q_d)) = k\}$. 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. Given a test query q and history $f_{0:t-1}$, the next-tool prediction in DTDR-C can be obtained with $\mathcal{F}_t = \omega(q, f_{0:t-1}) = G_{C(\theta(q))}(f_{0:t-1})$, or, with simpler notation, $\mathcal{F}_t = G_{C,q}(f_{0:t-1})$. The number of learned parameters in the K-means model is $e_\theta * K$, only depending on the output size of the embedding model e_θ and the size of the number of clusters K.

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 θ to predict the next function given both the query and the function trajectory until t. Formally, $\phi(\theta(q + f_{0:t-1}))$, where the operator $+$ denotes concatenation between strings. We use a softmax operation to obtain a probability distribution from the mode outputs. Let $P_\phi(f|q, f_{0:t-1}))$ be the probability of f given the query and trajectory. To retrieve only a subset of tools, we set a threshold value α, so that $\mathcal{F}_t = \omega(q, f_{0:t-1}) = \{f|f \in \mathcal{F}, P_\phi(f|q, f_{0:t-1}) > \alpha\}$. The rows of the linear layer ϕ can be

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. 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.

	Retrieval Method	Function Selection Acc. [%] (↑)	Retrieval Metrics MRR (↑)	Retrieval Metrics F_1 (↑)
TinyAgent	Random Guess	5.9	0.26	0.12
	BM-25 (Robertson et al., 2009)	23.1	0.35	0.20
	QTS (Embeddings Similarity)	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)	1.0	0.18	0.05
	QTS (Embeddings Similarity)	2.1	0.21	0.08
	QTS (Gao et al., 2025)	9.9	0.37	0.27
	QTS (Paramanayakam et al., 2025)	0.0	0.30	0.18
	DR (Liu et al., 2024a)	3.7	0.46	0.33
	Dynamic DR (DTDR-C) (Ours)	7.6	0.64	0.55
	LR (Erdogan et al., 2024)	12.8	0.53	0.32
	Dynamic LR (DTDR-L) (Ours)	**13.8**	**0.75**	**0.63**
Taskbench-MM	Random Guess	2.4	0.11	0.03
	BM-25 (Robertson et al., 2009)	9.9	0.26	0.19
	QTS (Embeddings Similarity)	1.3	0.14	0.04
	QTS (Gao et al., 2025)	1.0	0.14	0.20
	QTS (Paramanayakam et al., 2025)	0.8	0.29	0.18
	DR (Liu et al., 2024a)	3.4	0.38	0.27
	Dynamic DR (DTDR-C) (Ours)	5.9	0.52	0.43
	LR (Erdogan et al., 2024)	**14.1**	0.49	0.28
	Dynamic LR (DTDR-L) (Ours)	13.8	**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 (Embeddings Similarity)	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**

interpreted as learned embeddings of the tools $\mathcal{F}$, which are trained to align with the query embeddings $\theta(q)$. The number of learned parameters in this model is $e_\theta * |\mathcal{F}|$, only depending on the output size of the embedding model e_θ and number of tools $|\mathcal{F}|$. More implementation details in Appendix C.

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 reset is used as demonstration trajectories for tool retrieval and ICL methods. In Figure 5c, we investigate decreasing the number of demonstrations.

LLM backbones. We evaluate tool-retrieval methods and ICL strategies applied to seven LLM backbones. We consider the *Qwen 3* family of models (0.6B, 1.7B, 4B, 8B, 14B) (Yang et al., 2025) represent-

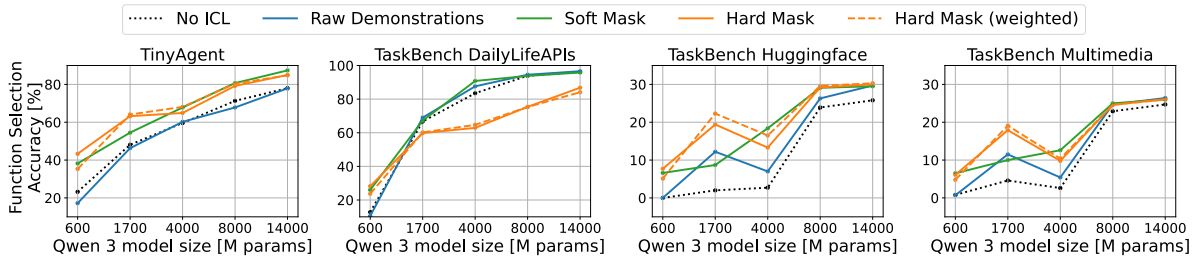

Figure 3: Comparison of efficient ICL methods against ICL with Raw Demonstrations and the baseline without ICL. All results are conditioned using the *K-means* retriever.

ing 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. We evaluate the methods across three dimensions: downstream performance, retrieval quality, and computational efficiency. 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 DAG of the ground-truth plan. For the *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 F. 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. We assess efficiency by reporting: *prompt length* as a proxy for LLM 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 Retrievers 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 Retrieval (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 *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). Additional implementation 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 subset of tools obtained from the Tool Retriever. For the *Raw Demonstrations* method, we provide up to 5 additional demonstrations for which the plan includes the predicted tools. For a 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. If a Tool Retriever also predicts scores for each tool, these can be included in the prompt, yielding *Weighted* variants of the two masking approaches. Example of ICL prompts are provided in Appendix F.

6 Results and Discussion

Tool Retrieval Methods. We evaluate Tool Retrievers on ranking, retrieval and downstream performance (see Figure 2). Higher MRR scores generally correlated well with the Function Selection Accuracy, indicating that better ranking simplifies LLM reasoning. Set-based metrics like F_1 score are less informative of the downstream performance, as they do not capture the order and score of tools,

Table 3: 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).

	Function Selection Accuracy [%] (↑)				
	Method	TinyAgent	TaskBench DailyLife	TaskBench HuggingFace	TaskBench Multimedia
Qwen3 0.6B	No ICL	23.2	12.8	0.0	0.8
	Raw Demonstrations	17.2	10.5	0.0	0.7
	Dependency Retrieval (Liu et al., 2024a)	22.0	11.5	1.6	2.3
	Learned Retrieval (Erdogan et al., 2024)	19.8	19.6	4.4	0.6
	DTDR-C (Ours)	35.3	23.7	5.1	4.8
	DTDR-L (Ours)	**46.1**	**45.9**	**6.0**	**6.9**
Qwen3 1.7B	No ICL	48.1	66.7	2.0	4.6
	Raw Demonstrations	46.3	69.0	12.2	11.5
	Dependency Retrieval (Liu et al., 2024a)	51.6	60.0	12.5	12.3
	Learned Retrieval (Erdogan et al., 2024)	50.5	56.6	**22.6**	**20.7**
	DTDR-C (Ours)	64.1	60.2	22.3	19.1
	DTDR-L (Ours)	**78.4**	**83.1**	21.4	18.3
Qwen3 4B	No ICL	59.6	83.5	2.7	2.6
	Raw Demonstrations	60.2	87.6	7.0	5.4
	Dependency Retrieval (Liu et al., 2024a)	62.7	64.7	13.0	7.3
	Learned Retrieval (Erdogan et al., 2024)	52.2	60.4	**30.5**	**25.9**
	DTDR-C (Ours)	68.1	64.7	16.5	10.4
	DTDR-L (Ours)	**80.7**	**89.0**	28.5	25.4
Qwen3 8B	No ICL	71.3	93.9	23.9	22.9
	Raw Demonstrations	67.8	**94.5**	26.3	24.7
	Dependency Retrieval (Liu et al., 2024a)	76.7	75.3	28.8	23.6
	Learned Retrieval (Erdogan et al., 2024)	43.7	52.2	**31.3**	**27.1**
	DTDR-C (Ours)	80.4	75.3	29.6	24.6
	DTDR-L (Ours)	**89.0**	91.0	30.3	26.8
GPT-4o	No ICL	84.7	96.6	28.5	24.4
	Raw Demonstrations	86.8	**96.8**	30.2	25.7
	Dependency Retrieval (Liu et al., 2024a)	85.9	82.9	29.4	25.3
	Learned Retrieval (Erdogan et al., 2024)	53.2	48.1	**31.3**	**27.0**
	DTDR-C (Ours)	85.9	82.9	30.4	26.0
	DTDR-L (Ours)	**94.2**	91.5	30.8	**27.0**

which is important when applying the thresholding function. Overall, *Learned Retriever* and *Dependency Retriever* methods perform better than other baselines. Our *dynamic* Learned Retriever DTDR-L surpasses the *static* counterpart (Erdogan et al., 2024) by over 35% on TinyAgent and TaskBench DL, while matching performance on TaskBench HF and MM, demonstrating the benefit of conditioning the prediction on the function selection history. Similarly, DTDR-C improves over the *static* Dependency Retriever (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. Surprisingly, the term-similarity baseline BM-25 remains competitive with *Embedding Similarity* methods. These methods compare query and tool descriptions. Since the "end-of-plan" function is not explicitly described in the query, it is never predicted, which occasionally results in retrieval scores worse than random. In terms of efficiency, BM-25 and the DR baseline are model-free; other methods use a small sentence encoder to represent the query, whose cost is negligible relative to the subsequent LLM prompting step. The only exception is Less-is-More Lv1, which additionally prompts the LLM to describe the ideal tool set for the task.

ICL Methods. In Figure 3, we analyze ICL strategies using tools retrieved with DTDR-C. Adding demonstrations mainly benefits domain-specific datasets like TaskBench HF and MM, where tool names and descriptions are more ambiguous. On three of four datasets, both Hard and Soft Masking outperform No ICL and Raw Demonstration baselines. The exception is TaskBench DL, which lacks intrinsic function-call dependencies (Table 5), making dependency-based prompting less relevant. Soft Masking consistently matches or exceeds Raw Demonstrations, while Hard Masking achieves the highest accuracy for smaller models by directly simplifying the selection task. In contrast, Soft Masking only suggests tools, leaving the LLM to decide whether to trust them. Except for Qwen 3 0.6B, Weighted Hard Masking performs on par with or better than its unweighted variant. Despite the similar performance, when we investigate further their behavior we notice that the latter is better in specific cases where the retriever assigns low probabilities to all the ground-truth tools. This insight can help in scenarios with distribution shift between test queries and demonstration data, where no confident predictions can be

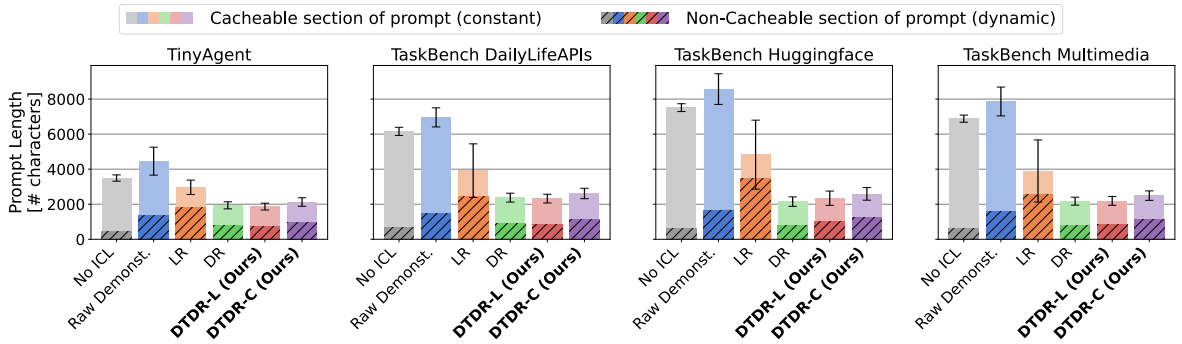

Figure 4: Prompt length across different methods and datasets. Our method reduces the prompt length by: 1) efficiently encoding ICL examples as tool dependencies instead of Raw Demonstrations, and 2) only retrieving tool dependencies which are relevant for the test query.

made. More details can be found in Appendix E. Weighted Soft Masking yields negligible gains and is omitted from the plot for clarity. From an efficiency standpoint, while Raw Demonstrations increase prompt length, Hard Masking reduces it substantially by removing descriptions of irrelevant tools.

Accuracy across LLM Backbones. We consolidate insights from previous experiments and evaluate selected methods for function selection across seven models and four datasets. In Table 3 (full table with Qwen 3 14B and Gorilla V2 in Appendix D), we report results using the best Tool Retrievers combined with weighted Hard Masking for ICL, alongside the Raw Demonstration baseline (demonstrations obtained via DTDR-C). Raw Demonstrations hurt smaller models but become competitive at larger scales, where LLMs exhibit stronger reasoning capabilities. DTDR-C consistently outperforms its static counterpart across all settings, while DTDR-L is comparable or better than the Learned Retrieval. This confirms that dynamic retrieval is key for high-quality function calling. Among all methods, DTDR-L achieves the best overall performance. Notably, for Qwen 3 4B and 8B, DTDR-L surpasses the No-ICL baseline of Qwen 3 14B and even GPT-4o, bridging the performance gap between edge and cloud models. At similar scales, 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 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.

Prompt Efficiency. In Figure 4, we analyze the same methods from a prompt-efficiency perspective. Since function selection requires generating only a few tokens, inference time is dominated by the prefill step. We therefore compare prompt lengths across methods and datasets, distinguishing between constant and variable prompt sections (see examples in Appendix F). The constant section (guidelines, task demonstrations, and the tool descriptions) can be precomputed and cached, while the variable section (query encoding and ICL-specific text) must be recomputed for each query. For the No ICL baseline the function descriptions occupy most of the prompt, which further increases in size when adding Raw Demonstrations. This quickly scales up as the number of tools increases (e.g., TaskBench HF). Static Dependency Retrieval reduces this by selecting a subset of functions, but the subset changes at each step and must be updated dynamically in the prompt. While the Learned Retriever can select a narrow set of tools, it tends to overfit to retrieving the most frequent tool, resulting in low end-to-end performance (see Table. 4). Our DTDR variants go further: dynamic retrieval narrows the candidate set of tools significantly, reducing total prompt length by up to 51% and the variable portion by up to 72%. Combined with accuracy gains, these results highlight the dual benefit of DTDR: improved reasoning and enhanced prompt efficiency.

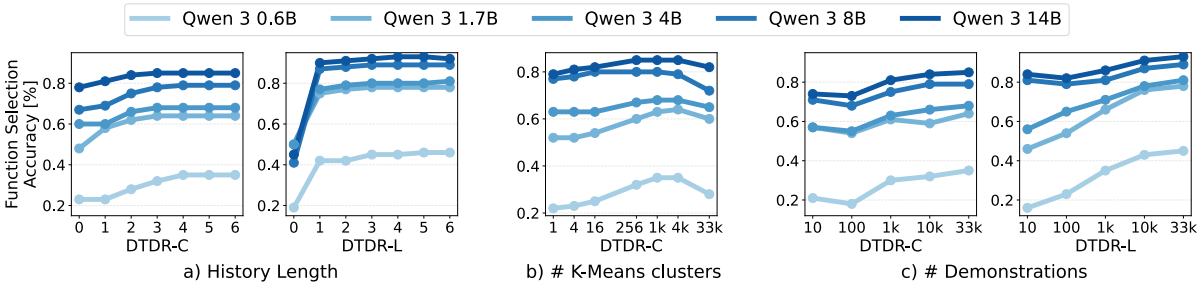

Figure 5: Ablations on: a) # of demonstrations, b) history length, and c) # of k-means clusters.

Table 4: End-to-end Success Rates for selected methods on compact edge LLMs. Best results per dataset are bolded. Across all small models and datasets, using a DTDR-based ω outperforms retrieval baselines.

	End-to-End Function Calling Success Rate (↑)				
	Method	TinyAgent	TaskBench DailyLife	TaskBench HuggingFace	TaskBench Multimedia
Qwen3 0.6B	No ICL	0.4	0.0	0.0	0.0
	Raw Demonstrations	0.1	0.0	0.0	0.0
	Dependency Retrieval (Liu et al., 2024a)	0.2	0.0	0.0	0.3
	Learned Retrieval (Erdogan et al., 2024)	0.9	2.0	0.5	0.7
	DTDR-C (Ours)	**4.2**	**6.8**	**5.9**	**8.3**
	DTDR-L (Ours)	3.5	0.8	1.2	3.7
Qwen3 1.7B	No ICL	4.4	13.5	3.6	4.9
	Raw Demonstrations	6.7	22.9	19.9	19.2
	Dependency Retrieval (Liu et al., 2024a)	2.6	9.7	5.7	6.8
	Learned Retrieval (Erdogan et al., 2024)	7.1	20.3	17.3	17.3
	DTDR-C (Ours)	11.8	26.4	**21.3**	**22.5**
	DTDR-L (Ours)	**14.5**	**29.4**	19.7	20.4
Qwen3 4B	No ICL	5.0	28.4	5.1	6.8
	Raw Demonstrations	3.6	26.2	10.7	9.8
	Dependency Retrieval (Liu et al., 2024a)	9.2	19.3	19.3	16.8
	Learned Retrieval (Erdogan et al., 2024)	9.0	19.3	28.0	26.5
	DTDR-C (Ours)	14.4	27.9	**35.1**	28.0
	DTDR-L (Ours)	**16.9**	**31.8**	29.8	**30.6**

End-to-End Results. In Table 4, we report end-to-end Success Rate for the function calling task, using the same model with different prompts for the function selection and parameter filling subtasks (details in Appendix F). If one or more mistakes occurs at any step in the plan (either in the 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. The Learned Retrieval (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. On the other hand, the clustering-based variant DTDR-C is based on the statistics of demonstration data, and does not have the issue of overfitting to most common functions. The baseline using Raw Demonstrations in the prompt reaches good performance with larger models, which have increased capabilities of reasoning on long text. However, as shown in Figure 4, using long demonstrations in the prompt makes the system latency significantly higher.

Impact of History Length, Clustering, and Number of Demonstrations. Figure 5a 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 5b 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 5c shows how increasing the number of demonstrations influences Function Selection Accuracy for both DTDR-C and DTDR-L. 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 demonstrations due to overfitting, highlighting its sensitivity compared to the more robust DTDR-C.

7 Conclusion and Future Work

We introduce Dynamic Tool Dependency Retrieval (DTDR), a method for task-aligned retrieval that captures multi-step tool dependencies. DTDR leverages both the query and historical predictions via a retrieval module trained on demonstrations, with two variants: a supervised approach and an unsupervised clustering-based approach, both suitable for low-resource, on-device settings. Experiments across multiple datasets and LLM backbones show DTDR significantly improves retrieval quality and downstream function selection over baselines. We further analyze prompt efficiency, history context, demonstration count, and clustering effects. Future directions include handling imperfect demonstrations, extending to multimodal tool-based tasks (e.g., robotics), and adapting to evolving tool sets.