In-N-Out: A Parameter-Level API Graph Dataset for Tool Agents

Seungkyu Lee[1]* **Nalim Kim**[2] **Yohan Jo**[2]†
[1]Department of Industrial Engineering, Seoul National University
[2]Graduate School of Data Science, Seoul National University

Abstract

Tool agents—LLM-based systems that interact with external APIs—offer a way to execute real-world tasks. However, as tasks become increasingly complex, these agents struggle to identify and call the correct APIs in the proper order. To tackle this problem, we investigate converting API documentation into a structured API graph that captures API dependencies and leveraging it for multi-tool queries that require compositional API calls. To support this, we introduce **In-N-Out**, the first expert-annotated dataset of API graphs built from two real-world API benchmarks and their documentation. Using In-N-Out significantly improves performance on both tool retrieval and multi-tool query generation, nearly doubling that of LLMs using documentation alone. Moreover, graphs generated by models fine-tuned on In-N-Out close 90% of this gap, showing that our dataset helps models learn to comprehend API documentation and parameter relationships. Our findings highlight the promise of using explicit API graphs for tool agents and the utility of In-N-Out as a valuable resource. We release our dataset and code at https://github.com/holi-lab/In-N-Out-API-Graph.

1 Introduction

Recent advances in Large Language Models (LLMs) have spurred growing interest in building *tool agents*—LLM-based systems that interact with external APIs to execute real-world tasks (Schick et al., 2023; Shen et al., 2023). Unlike traditional language models that rely solely on pretrained knowledge, tool agents can retrieve real-time information, access databases, and operate services via API calls (Song et al., 2023). As user queries grow more complex, solving them often involves chaining multiple APIs—commonly known as **multi-tool queries**—where one API call requires the outputs produced by a previous call. This makes it crucial for tool agents not only to understand individual APIs but also to capture interdependencies among APIs to plan valid call sequences (Lumer et al., 2025; Zhang et al., 2024).

A concrete example in Figure 1a illustrates why such interdependencies are essential. To fulfill the query *"Follow all the EDM artists on Spotify that have at least 1K followers"*, an agent must first call other APIs to obtain an access token and artist IDs, before invoking FollowArtist(). Without explicit knowledge of parameter connections, agents often fail to chain these APIs correctly, as the required dependencies are obscured among the large number of available APIs.

While prior studies have explored training tool agents on large datasets of multi-tool queries (Qin et al., 2024; Shim et al., 2025), a more natural and general solution, akin to what human developers do, is to use API documentation to identify API dependencies. However, since LLMs struggle to comprehend parameter-level dependencies from noisy, real-world API documentation (§4), we focus on a relatively underexplored approach: converting API documentation into an explicit API graph, enabling LLMs to use this structured information. Some prior studies that have attempted to use API graphs have focused primarily on limited domains (Wang et al., 2025b) or automatic graph construction with heuristics (Liu et al., 2024b; Shen et al., 2024), leaving the true potential of accurate API graphs in real-world scenarios unexplored. Since manual graph construction is not scalable for arbitrary API sets, a high-quality dataset is needed to train and evaluate a documentation-to-graph conversion module and ensure its generalizability to unseen APIs.

* This work was done while the first author was an intern at the Graduate School of Data Science at Seoul National University.
† Corresponding author.

Transactions of the Association for Computational Linguistics, vol. 14, pp. 1008–1027, 2026. https://doi.org/10.1162/TACL.a.694
Action Editor: Mihai Surdeanu. Submission batch: 9/2025; Revision batch: 11/2025; Published 6/2026.

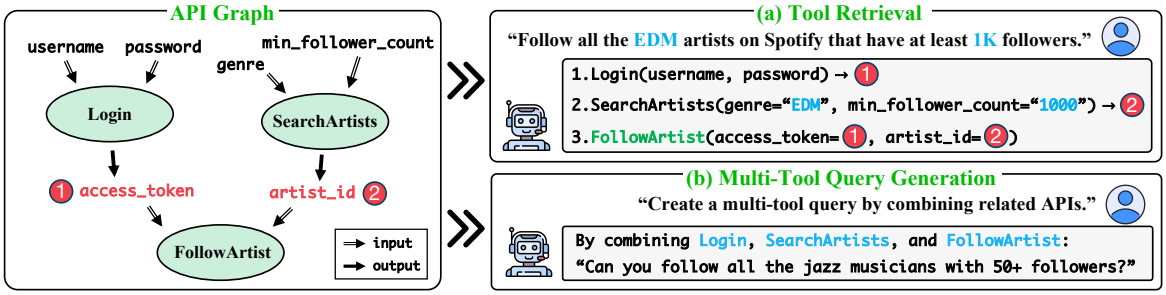


Figure 1: Illustration of how parameter-level API graphs support (a) **Tool Retrieval**, where the agent identifies prerequisite APIs needed to supply inputs for a target API (e.g., FollowArtist), and (b) **Multi-Tool Query Generation**, where the agent selects interdependent APIs to construct coherent multi-tool queries.

Our approach is arguably more generalizable and scalable than simply training a tool agent on large multi-tool tasks, and the graphs can even facilitate the generation of multi-tool queries by providing the underlying API composition for complex queries (Figure 1b).

Against this backdrop, we introduce **In-N-Out**, an API graph dataset constructed through expert annotation. In-N-Out represents both APIs and their parameters as nodes, with directed edges indicating when an output from one API can serve as a valid input to another. The APIs are sourced from AppWorld (Trivedi et al., 2024) and NESTful(v1) (Basu et al., 2025), spanning 550 APIs, and over 30,000 parameter-level edges, thus covering diverse and realistic tool-use scenarios.

We first demonstrate that training LLMs on our In-N-Out dataset improves their ability to construct parameter-level API graphs from noisy documentation, even for unseen APIs. Specifically, we prompt models to decide whether a connection exists between the parameters of two given APIs. Zero-shot models perform poorly on this task, indicating that LLM-based automatic graph construction is unreliable. In contrast, models fine-tuned on In-N-Out achieve significantly higher accuracy and generalize well to unseen APIs and datasets, sometimes reaching the accuracy achieved by training on the test dataset itself. This demonstrates that In-N-Out allows LLMs to learn the underlying ability to comprehend API documentation and parameter relationships.

We further draw two key insights from experiments on two tasks: (1) solving multi-tool queries and (2) generating multi-tool queries (which may be used for training and benchmarking tool agents). First, using In-N-Out significantly improves performance on both tasks compared to using documentation alone. This justifies our approach of using explicit, high-quality API graphs for overcoming challenges in tool agents. Second, even automatically generated graphs from models fine-tuned on In-N-Out yield substantial gains, closing over 90% of the performance gap on both tasks compared to using documentation directly. The results demonstrate that In-N-Out can serve as a valuable resource for graph-based approaches for tool agents.

In summary, our contributions are as follows:

- We construct and release In-N-Out, an API graph dataset annotated by experts, capturing parameter-level dependencies between real-world APIs.
- We show that fine-tuning on In-N-Out enables LLMs to infer parameter-level connections from textual API documentation and generalize to unseen APIs.
- We demonstrate that In-N-Out substantially improve tool agent performance, and that even automatically generated graphs reduce the performance gap relative to human-labeled graphs.

2 Related Work

Developing tool agents has centered on two key tasks: tool retrieval and planning (§2.1), and multi-tool query generation (§2.2). Graph-based approaches have supported both tasks by capturing API relationships and enabling scalable coordination.

2.1 Tool Retrieval and Planning

As real-world tasks increasingly require multiple APIs in sequence, both accurate retrieval and correct input provision become essential (Patil et al.,

2025; Xu et al., 2024; Shi et al., 2025; Du et al., 2024; Lumer et al., 2024). To support these capabilities, previous work has explored several approaches: refining API documentation to make it more interpretable for LLMs (Hsieh et al., 2023; Yuan et al., 2025; Chen et al., 2024), fine-tuning LLMs on query-API call pairs (Qin et al., 2024; Patil et al., 2024), and learning tool-specific embeddings (Hao et al., 2023; Wang et al., 2025a). However, these strategies often fail to capture the underlying dependencies among APIs, while also requiring high training costs and generalizing poorly to unseen APIs.

To address this, recent studies have explored graph-based approaches that represent API relationships explicitly to support tool retrieval (Liu et al., 2024b,a; Chen et al., 2025). GraphRAG-Tool Fusion (Lumer et al., 2025) constructs graphs by modeling both API- and parameter-level dependencies, but relies on synthetic APIs that lack the complexity of real-world environments. SoAy (Wang et al., 2025b) uses real APIs but is limited to academic information-seeking tasks with only seven APIs and simple parameter structures, falling short of the real-world requirement to combine many APIs across diverse domains. In contrast, we construct expert-verified graphs over real APIs across 25 domains, capturing richer relationships and enabling reliable tool retrieval in real-world settings.

2.2 Multi-Tool Query Generation

Another direction for developing tool agents is to fine-tune them on queries paired with the API call trajectories needed to resolve them, providing direct supervision for both planning and execution (Zhuang et al., 2023; Li et al., 2023; Huang et al., 2024). For such training to be effective, the queries must reflect genuine interdependencies among APIs (Shen et al., 2025; Wu et al., 2024). However, existing efforts face two fundamental challenges: ensuring validity and achieving scalability. NESTful(v1) (Basu et al., 2025) builds nested queries from inferred relationships with human validation, but its queries are limited in scale. NesTools (Han et al., 2025) address scalability by automatically generating queries with LLMs, but relies on synthetic APIs.

To overcome these issues, recent work has leveraged graph structures to systematically generate multi-tool queries (Wang et al., 2025b). TaskBench (Shen et al., 2024) constructs graphs by matching parameter data types (e.g., image, text), and samples subgraphs to generate queries. ToolDial (Shim et al., 2025) connects APIs through keyword matching and embedding similarity, and generates queries by traversing pairs of connected nodes. While scalable, these approaches often connect semantically incompatible APIs or capture only coarse relations. In contrast, our work constructs parameter-level API graphs through expert annotation, providing accurate ground-truth connections that support valid query generation reflecting genuine dependencies. Moreover, models trained on these graphs can generalize to unseen APIs, enabling scalable inference of new dependencies and corresponding query generation.

3 In-N-Out Dataset

To construct a parameter-level API graph, we aim to capture cases where an output parameter of one API is compatible with an input of another. Inferring such dependencies with LLMs is unreliable, since API documentations often contain ambiguities, vague references, and inconsistent terminology (see Appendix A for examples). Alternatively, one might attempt to verify dependencies through direct API execution, but this is often infeasible due to deprecated or inaccessible APIs, and execution success or failure does not guarantee true semantic compatibility.

Therefore, we adopt the perspective of a developer reading documentation to infer relationships between APIs, and use this as the basis for ground truth. Two experienced software developers, each with over five years of programming experience and a background in computer science, independently review API documentation, assess intended usage, and decide whether a valid connection exists between API parameters.

Our annotations are built on two realistic and diverse datasets: NESTful(v1) (Basu et al., 2025) and AppWorld (Trivedi et al., 2024). NESTful contains 38 real-world APIs sourced from RapidAPI Hub[1] across 15 domains (e.g., weather, music, news), along with 85 multi-tool queries. We extend it with 55 additional APIs from the same domains to enrich the graph with more diverse relationships. AppWorld offers a simulated environment for 9 real-world apps (e.g., amazon, venmo,

[1] https://rapidapi.com/hub

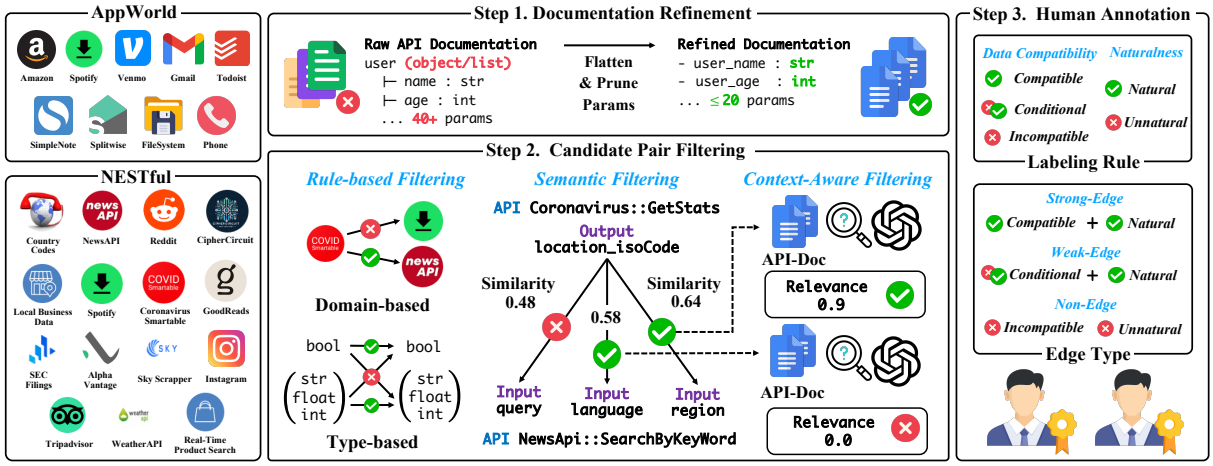


Figure 2: Construction process of the In-N-Out dataset: (1) refine API documentation, (2) filter candidate parameter pairs (rule-based, semantic, context-aware), (3) annotate edges by compatibility and naturalness.

spotify) with 457 APIs and 750 multi-tool tasks, providing a rich benchmark for compositional API usage. Based on these two datasets, we construct graphs that serve as a valuable resource for effective API composition.

We design a multi-stage pipeline to filter and annotate parameter-level connections between APIs. As shown in Figure 2, the overall process consists of three steps: documentation refinement (Step 1), candidate pair filtering (Step 2), and human annotation (Step 3).

3.1 Documentation Refinement

API documentation varies widely in format, and some return outputs through object or list structures that bundle multiple pieces of information. Because our goal is to capture parameter-level relationships between APIs, we first flatten such structured outputs into individual parameters and retain only primitive types: bool, str, float, and int.

Real-world APIs often produce a large number of output parameters, many of which are auxiliary values that are unlikely to be connected to other APIs, such as execution status flags, image URLs, or timestamps. Including such rare connections in the graph would add noise when a tool agent later uses it. To avoid this, we prune each API's outputs to at most 20 parameters, keeping only those that (1) support the API's core functionality and (2) are plausible inputs to other APIs in realistic tool-use contexts.

Moreover, because many APIs provide only output schema examples without parameter descriptions, we provide the full API documentation to the GPT-4o mini model and instruct it to generate a concise description (see Appendix E for the prompt), which will later be used in the candidate filtering stage.

3.2 Candidate Pair Filtering

Since the In-N-Out dataset defines edges at the parameter level, we first enumerate all possible pairs between output and input parameters. The number of initial pairs is extremely large: 279,928 for the 93 APIs in NESTful, and 1,916,351 for the 457 APIs in AppWorld. To reduce implausible connections and preserve the graph's utility, we apply three filtering processes—corresponding to Step 2 in Figure 2.

Rule-based Filtering. We eliminate pairs that are implausible either because their APIs belong to incompatible domains or because their types are mismatched. Domain-based filtering applies only to NESTful, which spans 15 diverse domains and thus produces irrelevant pairings such as combining a Coronavirus API with a Spotify API. Two experts independently reviewed the domain list and identified mutually incompatible domain pairs through agreement (see Appendix B for the irrelevant domain pairs). For type compatibility, we group the four primitive types into two categories: (1) Boolean values (bool) and (2) textual/numeric values (str, int, float), allowing connections only within the same category.

Semantic Filtering. To capture more subtle mismatches, we use SBERT (Reimers and Gurevych, 2019) to encode each parameter into an embedding based on its name and description, and filter pairs using cosine similarity. The similarity

threshold is set to 0.5, chosen to ensure that all API parameter pairs appearing in the annotated call sequences of NESTful and AppWorld are retained. After this step, 73,526 candidate pairs remain in NESTful and 428,040 in AppWorld.

Context-aware Filtering. Embedding similarity alone may still preserve pairs that look related lexically but differ in practical meaning (e.g., location_id vs. location_name). To filter these cases, we prompt GPT-4o mini with the full documentation of both APIs and ask it to score the relevance of each pair on a 0–1 scale (see Appendix E for the prompt). Here, relevance refers to whether two parameters would reasonably be linked in real tool-use scenarios, beyond type or lexical similarity. Using the same ground-truth pairs described above to guide the cutoff, we set the threshold to 0.3, which retains all gold connections. After this step, 3,066 candidate pairs remain in NESTful and 48,757 in AppWorld.

To validate the filtering pipeline, we randomly inspected 100 discarded pairs from each dataset. In both NESTful and AppWorld, none of the sampled pairs represented connections that a developer would realistically use, suggesting that our filtering process effectively removes implausible edges while preserving the connections most useful for subsequent annotation and graph construction. (see Appendix D for the filtered-out pairs).

3.3 Human Annotation

To obtain reliable ground-truth labels, we conducted a human annotation process with the two experts introduced earlier on the parameter pairs that remained after filtering. Each candidate pair was evaluated according to two criteria: data compatibility and naturalness.

Data compatibility measures whether an output from one API can serve as a valid input for another. Pairs are labeled as *compatible* if the output consistently provides the information required by the input (e.g., user_id from one API used as user_id in another). They are labeled as *incompatible* if the output cannot fulfill the input requirement under any circumstance (e.g., an image URL cannot serve as a date input). The intermediate case, *conditional*, applies if it depends on the content. For example, SimpleNote::ShowNote().content might contain an email address in some cases but not always, meaning the connection is possible but not guaranteed.

Naturalness evaluates whether the connection reflects realistic tool usage. For instance, ResetPassword() requiring a user's own email address is natural, but obtaining it through SearchFriends() would be contextually unnatural and deviated from typical API usage patterns, despite being technically possible. Pairs based on this criterion are labeled as *natural* or *unnatural*.

Before full-scale annotation, both annotators jointly labeled 100 parameter pairs to calibrate their understanding—ensuring a shared interpretations of the two criteria. Following this calibration, each annotator independently read the complete API documentation, including parameter names and descriptions, examined actual API call results if needed, and assigned labels. Pairs with any disagreement between the two annotators—9,271 pairs in AppWorld and 870 in NESTful—were resolved through discussion to produce final labels.

Based on the data compatibility and naturalness, each parameter pair was assigned to one of three edge types.

Strong-Edge occurs when the parameters are labeled as both compatible and natural, indicating a reliable and contextually appropriate data flow (e.g., Amazon::Login().access_token → Amazon::ShowOrder().access_token).

Weak-Edge corresponds to cases labeled as conditional and natural, where the connection is plausible but depends on certain conditions being met (e.g., SimpleNote::ShowNote().content → Gmail::SendEmail().email_addresses).

Non-Edge is assigned to any pair involving incompatible or unnatural labels, reflecting parameter pairs that are not semantically aligned or executable in practice (e.g., Venmo::SearchUsers().last_name → Phone::UpdateContact().email).

This process yields API graphs that reflect realistic and contextually valid parameter-level data flows for tool-agent reasoning.

3.4 Data Statistics

Table 1 presents the final annotation statistics. In the NESTful dataset, we identify 1,011 strong edges and 928 weak edges, together accounting for only 0.7% of the 279,928 possible parameter pairs. In AppWorld, 15,623 strong edges and 17,231 weak edges constitute just 1.7% of the

Dataset	Total Pairs	Annotated	Strong	Weak
NESTful	279,928	3,066	1,011	928
AppWorld	1,916,351	48,757	15,623	17,231

Table 1: Statistics in the In-N-Out dataset: total parameter pairs, filtered human-annotated pairs, and resulting strong/weak edge counts.

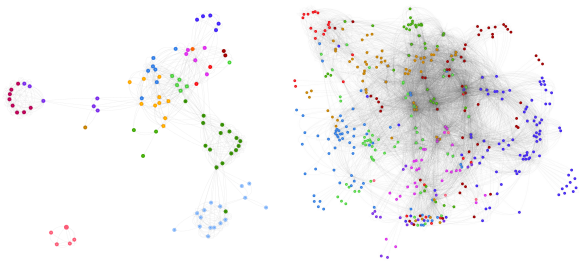
Connection Level	AppWorld		NESTful	
	d_{avg}	Cross	d_{avg}	Cross
Param → Param	22	78%	7	63%
API → API	32	76%	12	64%

Table 2: Connectivity statistics of In-N-Out. d_{avg} is the average number of incoming edges: (1) output parameters from other APIs linked to each input parameter (parameter level), and (2) upstream APIs providing inputs to each API (API level). Cross denotes the percentage of edges that cross domain boundaries.

1,916,351 possible pairs. All remaining parameter pairs are marked as non-edges. This extreme sparsity highlights the difficulty of discovering compatible parameter connections without pre-constructed resources such as an API graph.

As illustrated in Figure 3 and summarized in Table 2, the two datasets exhibit distinct connectivity patterns. In AppWorld, each API is connected to 32 other APIs on average, with 76% of connections spanning across domains. In contrast, APIs in NESTful are connected to only 12 other APIs on average, with 64% of them being cross-domain. At the parameter level, AppWorld also demonstrates relatively denser cross-domain links, whereas NESTful presents a more fragmented structure. By encompassing both styles, the In-N-Out dataset captures two representative patterns of API connectivity—one densely interconnected and the other more sparse—thus providing a robust resource for training and evaluating tool agents under diverse domain structures.

Although our graphs contain more cross-domain than in-domain connections (see the Cross column in Table 2), such multi-domain compositions are far less frequent in the original queries accompanied by NESTful and AppWorld. For example, only 22% of NESTful queries and 30% of AppWorld queries involve chaining APIs across different domains. This indicates that, without an explicit resource such as an API graph, it is particularly difficult to generate such cross-domain queries. Moreover, as we will show in the experiment section, agents can usually identify the prerequisite APIs needed for a target API call within the same domain, but their accuracy drops sharply across domains (around 80% in-domain for both datasets vs. 38% and 6% cross-domain for NESTful and AppWorld, respectively). These findings highlight cross-domain reasoning as a major bottleneck for tool agents, and demonstrate that explicit parameter-level graphs are essential for guiding them toward the correct dependencies.



(a) NESTful (b) AppWorld

Figure 3: Gold API Graphs for (a) NESTful and (b) AppWorld datasets. Nodes with the same color belong to the same domain.

4 Experiments

Our experiments address two main questions: (1) can LLMs construct parameter-level API graphs comparable to human annotations, and (2) do such graphs improve downstream tool-agent performance? To answer these, we first benchmark LLMs on graph construction with and without fine-tuning on In-N-Out (§4.1). We then evaluate whether explicit API graphs help agents retrieve prerequisite APIs that supply inputs to another (§4.2), and select interdependent API subsets to support multi-tool query generation (§4.3). We compare using gold graphs versus graphs generated by the fine-tuned LLM, quantifying the remaining gap due to imperfect graph construction. Finally, we conduct an end-to-end evaluation on τ-bench (§4.4) to examine whether the retrieval-level improvements observed in §4.2 translate into better task completion in realistic, multi-turn tool-use settings.

4.1 Benchmarking API Graph Construction Capabilities

We first measure the zero-shot performance of various LLMs on the parameter-level API graph construction task, and then examine how much these models improve after fine-tuning on the In-

N-Out. Since we build graphs for two distinct API datasets (NESTful and AppWorld), we also evaluate whether fine-tuning on one dataset generalizes effectively to the other.

Task Formulation. Given the full documentation $\mathcal{D}_A, \mathcal{D}_B$ of API A and API B, along with a specific output parameter p_{out} from A and an input parameter p_{in} from B, the model predicts the edge type—strong, weak, or non-edge:

$$f(\mathcal{D}_A, \mathcal{D}_B, p_{\text{out}}, p_{\text{in}}) \in \{\text{strong, weak, non}\} \quad (1)$$

To increase task difficulty, we evaluate only parameter pairs that pass through all three filtering stages. From each dataset, we allocate 300 pairs for validation and 300 pairs for testing (100 per class in each split), and use the remaining pairs for training (2,466 pairs for NESTful and 48,157 for AppWorld). For a more rigorous evaluation, we ensure that APIs from two specific domains are excluded from training and included only in the test set.

Comparison Models. We evaluate three categories of models: (1) closed-source zero-shot LLMs—GPT-4o mini, GPT-4o, GPT-4.1 mini, GPT-4.1; (2) open-source zero-shot LLMs—Llama-3.2-3B, Llama-3.1-8B, Qwen2.5-7B, Qwen2.5-32B; (3) fine-tuned versions of the open-source models, trained on In-N-Out. We report both in-dataset results—training and evaluation on the same dataset—and cross-dataset results—training on one dataset and evaluating on the other. Detailed training settings are provided in Appendix C.

Results. Table 3 summarizes the performance of all evaluated models. In the zero-shot setting, even high-performing closed-source models like GPT-4.1 reach only 70.0% on NESTful and 47.7% on AppWorld. Open-source models perform worse, with Qwen2.5-32B achieving 57.3% on NESTful and 50.3% on AppWorld. These results show that, despite their general reasoning ability, LLMs struggle to infer parameter-level API connections from documentation alone, highlighting the limitations of using zero-shot models for API graph construction.

Fine-tuning on In-N-Out yields large gains for all open-source models. Qwen2.5-32B reaches 76.3% on NESTful and 94.7% on AppWorld, while the smaller Llama-3.2-3B jumps from 39.0% to 76.7% and from 41.3% to 93.0%. These gains are achieved even though the test sets contain APIs from unseen domains. Notably, fine-tuning on AppWorld yields greater gains overall, likely because its graph contains nearly four times more APIs and over fifteen times more edges than NESTful—exposing models to a richer variety of parameter dependencies during training.

Model Type	Model	Accuracy (%)	
		NESTful	AppWorld
Closed-source	GPT-4o mini	56.0	44.7
	GPT-4o	61.0	42.3
	GPT-4.1 mini	53.7	**52.3**
	GPT-4.1	**70.0**	47.7
Open-source	Llama-3.2-3B	39.0	41.3
	Llama-3.1-8B	37.0	30.0
	Qwen2.5-7B	47.7	39.7
	Qwen2.5-32B	**57.3**	**50.3**
Fine-tuned (in-dataset)	Llama-3.2-3B	**76.7**	93.0
	Llama-3.1-8B	76.0	94.0
	Qwen2.5-7B	74.0	91.0
	Qwen2.5-32B	76.3	**94.7**
Fine-tuned (cross-dataset)	Llama-3.2-3B	72.3	52.3
	Llama-3.1-8B	70.7	61.3
	Qwen2.5-7B	71.7	55.7
	Qwen2.5-32B	**74.3**	**66.3**

Table 3: Performance of different models in predicting parameter-level edge types in the API graph construction task across NESTful and AppWorld datasets.

Cross-dataset experiments further confirm this generality. Qwen2.5-32B trained on NESTful and tested on AppWorld improves from 50.3% to 66.3%, surpassing the best zero-shot model. In the reverse setting, performance rises from 57.3% to 74.3%, nearly matching in-dataset fine-tuning. These results show that In-N-Out fine-tuning teaches general principles of parameter dependencies rather than dataset-specific patterns. We attribute this transferability to In-N-Out's diverse coverage, enabling models to extend effectively to entirely new APIs beyond those in training.

Analysis. To further probe model limitations, we analyzed fine-tuned Qwen2.5-32B's predictions in the cross-dataset setting, this time evaluating the model on the entire set of human-annotated parameter pairs. Specifically, we predicted edge types for all 48,757 annotated pairs in AppWorld and 3,066 in NESTful. The resulting automated graphs achieved 71.2% accuracy on AppWorld and 71.5% on NESTful. Since the labels are balanced across strong, weak, and non-edge classes, this evaluation reveals detailed error patterns.

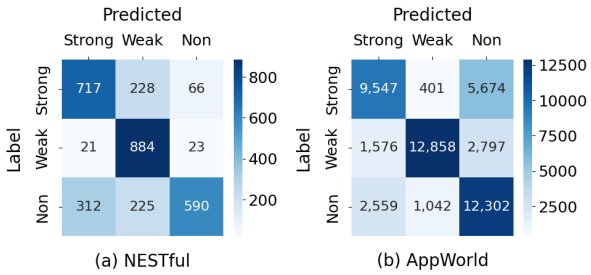


Figure 4: Confusion matrices of edge classification results with (a) NESTful, (b) AppWorld dataset.

Figure 4 shows contrasting error trends across datasets. In AppWorld, the model tends to underpredict strong dependencies—misclassifying 46% of cross-domain strong edges as non-edges, compared to 18% for in-domain edges (e.g., Amazon::ShowOrder().delivery_fee → Venmo::ShowTransactions().min_amount). In constrast, the model over-predicts in NESTful, labeling 48% of in-domain and 24% of cross-domain non-edges as strong (e.g., SkyScrapper::SearchAirport().entity_id → SkyScrapper::GetPriceCalendar().sky_id). These findings suggest that AppWorld errors mainly stem from difficulty in aligning semantically equivalent parameters across domains, whereas NESTful errors reflect over-reliance on lexical or structural similarity within a domain. Addressing these complementary failure modes will be key to improving the accuracy of automated graph construction.

4.2 Tool Retrieval with API Graphs

In realistic tool-use scenarios, solving a complex task often requires invoking multiple APIs in sequence, where each call may depend on outputs from earlier ones (Figure 1a). A key challenge is to identify which API can supply the missing input to a target API. This experiment evaluates whether API graphs from In-N-Out improve this retrieval process compared to a baseline without structural guidance.

Task Formulation. Given a task query q and the documentation $\mathcal{D}_A$ of a target API A with input parameters $\mathcal{P}(A)$, let $\mathcal{P}_{\text{miss}}(A) \subseteq \mathcal{P}(A)$ denote those parameters whose values are not explicitly specified in q. For each $p \in \mathcal{P}_{\text{miss}}(A)$, the objective is to select a prerequisite API B whose output parameter can supply the value for p:

$$f(q, \mathcal{D}_A, p) \rightarrow B. \quad (2)$$

To simulate this process, we compute SBERT embeddings (Reimers and Gurevych, 2019) for all candidate APIs and rank them by cosine similarity to the description of p, as LLM-based ranking over the full API set is impractical due to context length limits. When an API graph is available, we re-rank candidates by first promoting graph-connected APIs to the top (preserving their similarity-based order), followed by the remaining unconnected ones. This strategy ensures that graph structure informs the ranking without discarding semantic relevance. The top-5 ranked candidates are then passed to GPT-4o mini, which selects the final API B (see Appendix E for the prompt).

Evaluation instances are derived from task queries with gold API call sequences in the datasets. Each dependency where a target API requires an input provided by another API counts as a single instance, yielding 134 instances for NESTful. For AppWorld, 165 training and dev queries include gold sequences, but the 585 test queries do not. To obtain labels for these, we prompt GPT-4.1 to generate candidate sequences, execute them in the AppWorld simulator, and retain the 345 sequences that produce correct outputs (see Appendix E for the prompt). These were then validated by human annotators, resulting in 3,801 instances in total.

Performance is measured using: (1) Average Rank and Worst Rank of the correct API among all candidates ranked by embedding similarity; (2) Top-k Accuracy for $k \in \{1, 2, 5, 10, 20\}$, i.e., the proportion of instances where the correct API appears among the top k; (3) Final Selection Accuracy, where GPT-4o mini chooses the correct API from the top-5 list.

Comparison Methods. We compare three retrieval settings. In the **no-graph** setting, all APIs are ranked purely by embedding similarity to the missing parameter description. The **gold graph** setting uses ground-truth edges from In-N-Out to guide re-ranking. The **automated graph** setting applies the same re-ranking strategy using edges predicted by a Qwen2.5-32B model that is fine-tuned on the other dataset (cross-dataset setup).

Results. Table 4 shows that incorporating API graphs significantly improves tool retrieval performance across both datasets. Notably, even graphs automatically constructed by fine-tuned Qwen2.5-

Dataset	Condition	Rank (Gold API)		Gold API included in Top-k (%)					Final Selection Accuracy (%)
		Avg.	Worst	Top-1	Top-2	Top-5	Top-10	Top-20	
NESTful	No-graph	3.1	13	43.3	56.0	83.6	97.8	100.0	68.7
	Automated Graph	1.8	10	79.9	81.3	92.5	100.0	100.0	79.1
	Gold Graph	**1.6**	**10**	**84.3**	**89.6**	**92.5**	**100.0**	**100.0**	**79.9**
AppWorld	No-graph	19.3	449	51.8	55.0	58.6	63.9	68.7	57.3
	Automated Graph	8.1	393	64.6	70.7	78.3	82.9	84.5	73.0
	Gold Graph	**4.5**	**393**	**75.4**	**81.1**	**88.7**	**94.0**	**95.7**	**82.8**

Table 4: Tool retrieval performance under three graph conditions on NESTful and AppWorld (using parameter descriptions as search queries).

32B yield substantial gains over the no-graph baseline, underscoring the value of structural information beyond embedding-based similarity.

On NESTful, rank-based metrics show a striking improvement when graphs are used. Compared to the no-graph baseline, the average rank of the correct API drops from 3.1 to 1.6 with the gold graph, and Top-1 accuracy rises from 43.3% to 84.3%, reducing the gap by over 40 percentage points. This confirms that retrieval is far more effective than embedding-based similarity alone once explicit graph structure is introduced. Moreover, even with automatically constructed graphs, performance closely approaches the gold graph; accordingly, Final Selection Accuracy also improves substantially, from 68.7% (no-graph) to 79.1% (automated) and 79.9% (gold).

In AppWorld, where the search space is much larger, the gains are even more pronounced. The average rank of the correct API improves from 19.3 without a graph to 4.5 with the gold graph, while Top-1 accuracy rises from 51.8% to 75.4%. Automated graphs again close much of this gap, reaching 8.1 average rank and 64.6% Top-1 accuracy. Final Selection Accuracy also increases markedly, from 57.3% (no-graph) to 73.0% (automated) and 82.8% (gold).

Overall, even imperfect automated graphs—constructed under the challenging cross-dataset setting—consistently outperform the no-graph baseline by re-ranking candidates so that the most relevant APIs appear near the top. These results highlight that more accurate graph construction would directly translate into further gains in retrieval accuracy, making graph-based retrieval even more powerful for tool agents.

4.3 Structured API Subset Selection for Multi-Tool Query Generation

Generating multi-tool queries is essential not only for benchmarking but also for training models to solve complex tasks involving multiple APIs. A critical first step is to select subsets of APIs that can be composed into valid sequences via their input-output dependencies (Figure 1b). Here, we evaluate whether the In-N-Out graph helps models select such subsets more effectively. Specifically, we compare model performance with and without access to the graph, measuring how accurately LLMs identify API subsets that satisfy predefined structural patterns and target subset sizes.

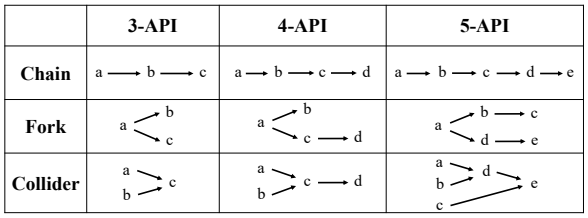


Figure 5: Predefined structural patterns (Chain, Fork, Collider) illustrated for 3-, 4-, and 5-API cases.

Task Formulation. Given a set of APIs $\mathcal{A}$ with documentation $\mathcal{D}_{\mathcal{A}}$, the model must generate exactly five candidate subsets $\mathcal{S} = \{S_1, S_2, \ldots, S_5\}$. Each S_i contains $n \in \{3, 4, 5\}$ APIs whose parameter dependencies form a target structural pattern $\pi \in \Pi = \{\text{Chain}, \text{Fork}, \text{Collider}\}$ (Figure 5). Formally, the task can be defined as:

$$f(\mathcal{D}_{\mathcal{A}}, n, \pi) \to \mathcal{S}, \quad |S_i| = n, \ |\mathcal{S}| = 5. \qquad (3)$$

For the 3-, 4-, and 5-API cases, $\mathcal{A}$ contains 15, 20, and 25 APIs respectively—sized to ensure that at least five valid subsets of the target pattern can be formed, while keeping the search space sufficiently challenging. Since additional valid connections may exist beyond the guaranteed ones, a generated subset is considered correct if it both satisfies the structural pattern and appears as a valid subgraph in the gold graph. Each configuration is sampled 100 times, and results are averaged across runs.

Dataset	# APIs	Condition	Chain	Fork	Collider
NESTful	3-API	No-graph	58.2	38.6	38.8
		Automated Graph	86.4	70.0	64.6
		Gold Graph	**90.2**	**76.2**	**67.2**
	4-API	No-graph	49.8	19.6	21.8
		Automated Graph	76.6	23.6	41.2
		Gold Graph	**84.0**	**34.4**	**43.8**
	5-API	No-graph	37.2	17.4	6.4
		Automated Graph	60.4	22.2	11.6
		Gold Graph	**68.0**	**22.2**	**17.4**
AppWorld	3-API	No-graph	41.2	32.2	23.8
		Automated Graph	71.8	70.6	59.8
		Gold Graph	**81.2**	**72.6**	**68.0**
	4-API	No-graph	35.6	16.0	15.6
		Automated Graph	47.0	34.0	41.0
		Gold Graph	**60.8**	**40.2**	**46.4**
	5-API	No-graph	5.6	13.0	6.2
		Automated Graph	22.8	24.6	13.6
		Gold Graph	**30.6**	**26.8**	**23.4**

Table 5: Precision (%) across API structure sizes and patterns under three graph conditions, for both NESTful and AppWorld.

Performance is measured by precision: the proportion of generated subsets S_i that meet the target pattern and appear in the gold graph, with duplicates counted once. As the model always outputs five subsets per instance, precision alone reliably reflects performance, without the need for recall.

Comparison Methods. We use GPT-4o mini as the base model and evaluate three conditions from §4.2. In the **no-graph** setting, the model selects subsets using only API documentation, with no structural guidance. In the **gold graph** setting, we additionally provide the ground-truth connections between the given APIs as part of the task instruction. In the **automated graph** setting, the same type of connection information is provided using edges predicted by the Qwen2.5-32B model fine-tuned on the other dataset (cross-dataset setup). Prompt details are provided in Appendix E.

Results. The results in Table 5 show that access to graph information significantly improves precision across structural patterns and subset sizes. For instance, in the 3-API chain setting, precision rises from 58.2% to 90.2% in NESTful and from 41.2% to 81.2% in AppWorld. Even in more challenging settings like the 5-API collider pattern, performance rises from 6.4% to 17.4% in NESTful and from 6.2% to 23.4% in AppWorld. These results demonstrate that gold graphs provide strong structural guidance that greatly enhances subset selection.

Automated graphs recover a large portion of this performance gap. In the AppWorld 3-API fork case, for example, precision reaches 70.6% with the automated graph, recovering over 95% of the gain achieved by the gold graph's 72.6%. Across conditions, automated graphs generally recover 70–90% of the performance improvements provided by gold graphs, demonstrating that even imperfect structural cues can effectively guide LLMs in extracting valid API subsets.

When comparing datasets, we find that gold graphs yield consistently larger performance gains in AppWorld than in NESTful. This indicates that structural cues are especially helpful in more complex environments—like AppWorld, with denser connections and more cross-domain edges. As API sets grow larger and more entangled, graph-based supervision becomes increasingly essential for identifying valid API subsets beyond what documentation alone can support. Still, models struggle with more complex patterns (e.g., 5-API collider), underscoring the need to improve their ability to fully utilize structural information during compositional reasoning.

4.4 End-to-End Performance Evaluation

While our earlier experiments (§4.2) demonstrate that API graphs substantially improve the retrieval of prerequisite APIs, this raises the question of whether such improvements translate into better end-to-end agent performance. To examine this, we conduct an additional evaluation using the τ-bench (Yao et al., 2025), which evaluates multi-turn tool use in realistic user–agent dialogues.

Setup. τ-bench consists of two domains, τ-retail and τ-airline, containing 15 and 13 API tools, respectively, and 115 and 50 test instances. Since the API documentation in τ-bench does not include output parameter information, we manually created extended documentation by inspecting each API's invocation results. From these, we enumerated all possible input–output parameter pairs—3,673 for τ-retail and 7,033 for τ-airline.

To construct API graphs, we used the Qwen2.5-32B model fine-tuned on the AppWorld portion of In-N-Out to predict edge types for all parameter pairs and retained only those classified as *strong* edges. The resulting graphs contained 182 edges for τ-retail and 537 edges for τ-airline. Using these graphs, we augmented each API's documentation by appending prerequisite API information to the corresponding input parameter descriptions.

This allows the agent to reference dependency-aware documentation during tool planning and execution.

Comparison Methods. Following the experimental setup of τ-bench, we adopted the tool calling agent described in the original paper as our base implementation. This agent uses function calling mode, where the model autonomously decides at each turn whether to respond to the user or invoke a tool. We compare two settings: the **baseline**, which uses the original τ-bench documentation, and the **automated graph** setting, where prerequisite API information predicted from our constructed graphs are incorporated into each tool's documentation. Both GPT-4o-mini and GPT-4o are used as the underlying reasoning models.

Results. Table 6 summarizes the end-to-end task accuracy across both domains. Overall, incorporating graph information led to consistent improvements over the baseline, with the largest gain observed on τ-airline using GPT-4o (from 50% to 60%). This indicates that structured dependency cues can help higher-capacity models utilize prerequisite information more effectively during tool planning.

Notably, when we qualitatively inspected the agent's tool-calling trajectories, we observed that the graph-augmented documentation helped the agent correctly identify and invoke prerequisite APIs capable of supplying missing input values for subsequent calls. For instance, in τ-airline baseline setting, the agent originally hallucinated a passenger's *dob* (date of birth) by substituting it with the flight's departure date. After we added to the documentation that *get_user_details* provides the required *dob* parameter, the agent correctly called *get_user_details* first and then used the retrieved date in the booking API, demonstrating the utility of structural guidance.

Discussion. Although graph augmentation led to measurable improvements, the overall gains were modest—a result that is closely reflects the inherent challenges of τ-bench. According to the failure breakdown reported in the original paper, 56% of errors stem from using wrong arguments or information, 25% from incorrect decisions, and 19% from partially resolved compound requests. Among these, only the first category can be partially addressed by the dependency structure encoded in our API graphs, whereas decision errors or domain-rule violations are largely orthogonal to graph-based reasoning. Moreover, a single task in τ-bench can exhibit multiple failure types simultaneously, making overall end-to-end improvements difficult to achieve.

Model	Condition	Accuracy (%)	
		τ-airline	τ-retail
GPT-4o-mini	Baseline	20.0	37.4
	Automated Graph	**26.0**	**40.0**
GPT-4o	Baseline	50.0	60.9
	Automated Graph	**60.0**	**62.6**

Table 6: End-to-end task accuracy on τ-bench. Results are reported for the tool calling agent using original documentation (Baseline) versus graph-augmented documentation (Automated Graph).

Nevertheless, these findings suggest that while API graphs alone cannot resolve all reasoning failures, they meaningfully enhance the agent's ability to provide correct inputs for dependent API calls. Future work may explore more sophisticated ways of integrating API graphs into the agent's planning and dialogue process, beyond static documentation augmentation.

5 Conclusion

In this paper, we introduce In-N-Out, a parameter-level API graph dataset that captures realistic input–output dependencies across APIs. By explicitly encoding these connections, our graphs help tool agents move beyond superficial heuristics and reason about how APIs can be composed. Experiments show that fine-tuning LLMs on In-N-Out significantly improves their ability to convert documentation into structured API graphs, narrowing the gap with expert-built graphs. As a result, agents equipped with In-N-Out achieve substantial gains in tool retrieval and structured API subset selection, demonstrating that explicit graph structure is a powerful lever for advancing tool-agent performance. We believe this structure could also enable future work on graph-based traversal strategies that plan optimal API sequences from given inputs to desired outputs.

6 Limitations

Our study has several limitations that suggest directions for future research. First, In-N-Out currently covers only two benchmark datasets. Expanding to broader domains would improve

generalizability—particularly in heterogeneous or rapidly evolving tool ecosystems.

Second, some of our filtering and annotation criteria—such as judging whether a connection feels *natural*—can be inherently subjective. While we aligned annotator criteria through joint calibration, future work could further decompose the notion of *naturalness* into more objective dimensions. For instance, checking whether a user's private information is being paired with another person's public data can help flag unnatural information flow. Likewise, checking whether a parameter is being used across unrelated API domains can reveal domain-inappropriate usage. Such dimensions can provide clearer and more reproducible labeling guidelines.

Finally, our construction pipeline involves multiple filtering stages and substantial manual annotation. As the number of parameter pairs grows, the time and sustained attention required from annotators become a significant bottleneck. Future research could incorporate LLM-assisted labeling—for example, generating example input/output values to help annotators reason more intuitively about parameter compatibility. Complementary strategies such as retrieving similar previously labeled pairs may further support consistent decisions in ambiguous cases. In addition, a continual-learning workflow could reduce manual effort over time: a model trained on In-N-Out could propose labels for new parameter pairs, humans would verify them, and the refined annotations would be incorporated back into training. These directions would enable scalable dataset construction over a larger set of parameter pairs, which in turn may narrow the performance gap between automated and gold graphs and facilitate more robust tool-agent development.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant (RS-2024-00333484) and by the Institute of Information & Communications Technology Planning & Evaluation (IITP) under the Leading Generative AI Human Resources Development grant (IITP-2026-RS-2024-00397085) and the grant (RS-2025-02215122, Development and Demonstration of Lightweight AI Model for Smart Homes), all of which are funded by the Korean government (MSIT). This work was also supported by the National Supercomputing Center with supercomputing resources including technical support (KSC 2024-CRE-0554, KSC-2025-CRE-0332).

References

Kinjal Basu, Ibrahim Abdelaziz, Kiran Kate, Mayank Agarwal, Maxwell Crouse, Yara Rizk, Kelsey Bradford, Asim Munawar, Sadhana Kumaravel, Saurabh Goyal, Xin Wang, Luis A. Lastras, and Pavan Kapanipathi. 2025. NESTFUL: A benchmark for evaluating LLMs on nested sequences of API calls. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 33526–33535, Suzhou, China. Association for Computational Linguistics.

Yanfei Chen, Jinsung Yoon, Devendra Singh Sachan, Qingze Wang, Vincent Cohen-Addad, Mohammadhossein Bateni, Chen-Yu Lee, and Tomas Pfister. 2024. Re-invoke: Tool invocation rewriting for zero-shot tool retrieval. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 4705–4726, Miami, Florida, USA. Association for Computational Linguistics.

Zhaoling Chen, Robert Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. 2025. LocAgent: Graph-guided LLM agents for code localization. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8697–8727, Vienna, Austria. Association for Computational Linguistics.

Yu Du, Fangyun Wei, and Hongyang Zhang. 2024. Anytool: Self-reflective, hierarchical agents for large-scale api calls.

Han Han, Tong Zhu, Xiang Zhang, MengSong Wu, Xiong Hao, and Wenliang Chen. 2025. NesTools: A dataset for evaluating nested tool learning abilities of large language models. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 9824–9844, Abu Dhabi, UAE. Association for Computational Linguistics.

Shibo Hao, Tianyang Liu, Zhen Wang, and Zhiting Hu. 2023. ToolkenGPT: Augmenting frozen

language models with massive tools via tool embeddings. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Cheng-Yu Hsieh, Si-An Chen, Chun-Liang Li, Yasuhisa Fujii, Alexander Ratner, Chen-Yu Lee, Ranjay Krishna, and Tomas Pfister. 2023. Tool documentation enables zero-shot tool-usage with large language models.

Yue Huang, Jiawen Shi, Yuan Li, Chenrui Fan, Siyuan Wu, Qihui Zhang, Yixin Liu, Pan Zhou, Yao Wan, Neil Zhenqiang Gong, and Lichao Sun. 2024. Metatool benchmark for large language models: Deciding whether to use tools and which to use. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023. API-bank: A comprehensive benchmark for tool-augmented LLMs. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, Singapore. Association for Computational Linguistics.

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. *CoRR*, abs/2403.00839.

Zhaoyang Liu, Zeqiang Lai, Zhangwei Gao, Erfei Cui, Ziheng Li, Xizhou Zhu, Lewei Lu, Qifeng Chen, Yu Qiao, Jifeng Dai, and Wenhai Wang. 2024b. Controlllm: Augment language models with tools by searching on graphs. In *ECCV (12)*, pages 89–105.

Elias Lumer, Pradeep Honaganahalli Basavaraju, Myles Mason, James A. Burke, and Vamse Kumar Subbiah. 2025. Graph rag-tool fusion.

Elias Lumer, Vamse Kumar Subbiah, James A. Burke, Pradeep Honaganahalli Basavaraju, and Austin Huber. 2024. Toolshed: Scale tool-equipped agents with advanced rag-tool fusion and tool knowledge bases.

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. In *Advances in Neural Information Processing Systems*, volume 37, pages 126544–126565. Curran Associates, Inc.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. Toolllm: Facilitating large language models to master 16000+ real-world apis. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

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 and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, Hong Kong, China. Association for Computational Linguistics.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Haiyang Shen, Yue Li, Desong Meng, Dongqi Cai, Sheng Qi, Li Zhang, Mengwei Xu, and Yun Ma. 2025. Shortcutsbench: A large-scale real-world benchmark for API-based agents. In *The Thirteenth International Conference on Learning Representations*.

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. In

Thirty-seventh Conference on Neural Information Processing Systems.

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. In *Advances in Neural Information Processing Systems*, volume 37, pages 4540–4574. Curran Associates, Inc.

Zhengliang Shi, Yuhan Wang, Lingyong Yan, Pengjie Ren, Shuaiqiang Wang, Dawei Yin, and Zhaochun Ren. 2025. Retrieval models aren't tool-savvy: Benchmarking tool retrieval for large language models. In *Findings of the Association for Computational Linguistics: ACL 2025*, Vienna, Austria. Association for Computational Linguistics.

Jeonghoon Shim, Gyuhyeon Seo, Cheongsu Lim, and Yohan Jo. 2025. Tooldial: Multi-turn dialogue generation method for tool-augmented language models. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Yifan Song, Weimin Xiong, Dawei Zhu, Wenhao Wu, Han Qian, Mingbo Song, Hailiang Huang, Cheng Li, Ke Wang, Rong Yao, Ye Tian, and Sujian Li. 2023. Restgpt: Connecting large language models with real-world restful apis.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. 2024. AppWorld: A controllable world of apps and people for benchmarking interactive coding agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 16022–16076, Bangkok, Thailand. Association for Computational Linguistics.

Renxi Wang, Xudong Han, Lei Ji, Shu Wang, Timothy Baldwin, and Haonan Li. 2025a. Toolgen: Unified tool retrieval and calling via generation. In *The Thirteenth International Conference on Learning Representations*.

Yuanchun Wang, Jifan Yu, Zijun Yao, Jing Zhang, Yuyang Xie, Shangqing Tu, Yiyang Fu, Youhe Feng, Jinkai Zhang, Jingyao Zhang, Bowen Huang, Yuanyao Li, Huihui Yuan, Lei Hou, Juanzi Li, and Jie Tang. 2025b. Soay: A solution-based llm api-using methodology for academic information seeking. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.1*, KDD '25, page 2660–2671, New York, NY, USA. Association for Computing Machinery.

Mengsong Wu, Tong Zhu, Han Han, Chuanyuan Tan, Xiang Zhang, and Wenliang Chen. 2024. Seal-tools: Self-instruct tool learning dataset for agent tuning and detailed benchmark. In *NLPCC (2)*, pages 372–384.

Qiancheng Xu, Yongqi Li, Heming Xia, and Wenjie Li. 2024. Enhancing tool retrieval with iterative feedback from large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2024, Miami, Florida, USA, November 12-16, 2024*, pages 9609–9619. Association for Computational Linguistics.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. 2025. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. In *The Thirteenth International Conference on Learning Representations*.

Siyu Yuan, Kaitao Song, Jiangjie Chen, Xu Tan, Yongliang Shen, Kan Ren, Dongsheng Li, and Deqing Yang. 2025. EASYTOOL: Enhancing LLM-based agents with concise tool instruction. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 951–972, Albuquerque, New Mexico. Association for Computational Linguistics.

Yinger Zhang, Hui Cai, Xierui Song, Yicheng Chen, Rui Sun, and Jing Zheng. 2024. Reverse chain: A generic-rule for LLMs to master multi-API planning. In *Findings of the Association for Computational Linguistics: NAACL 2024*, pages 302–325, Mexico City, Mexico. Association for Computational Linguistics.

Yuchen Zhuang, Yue Yu, Kuan Wang, Haotian Sun, and Chao Zhang. 2023. ToolQA: A

dataset for LLM question answering with external tools. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Appendix

A Examples of Ambiguous API Documentation

```
{
   "name": "Goodreads_GetBookByUrl",
   "description": "Get a book by its
   URL.",
   "query_parameters": [
      {
         "name": "bookURL",
         "type": "str",
         "description": "The URL of the
         book on Goodreads."
      }
   ],
   "output_parameters": [
      "author_id": "str",
      "author_legacyId": "int",
      ...
   ]
},
{
   "name": "Goodreads_GetAuthorsBooks",
   "description": "Retrieves books
   authored by a specific author.",
   "query_parameters": [
      {
         "name": "authorID",
         "type": "str",
         "description": "ID of the
         author whose books are to be
         retrieved. Can be obtained
         through other endpoints."
      },
      ...
   ],
   "output_parameters": [
      "bookId": "str",
      ...
   ]
}
```

Listing 1: Examples of ambiguous API documentation from NESTful dataset. Problematic parameters are highligheted in red.

In Listing 1, Goodreads_GetBookByUrl returns two identifiers (author_id, author_legacyId), while Goodreads_GetAuthorsBooks requires a single authorID. The documentation does not clarify whether authorID corresponds to one or both of the returned identifiers, making the intended mapping ambiguous.

```
{
   "app_name": "Phone",
   "api_name": "SearchContacts",
   "description": "Search your contact
   book for relatives' information.",
   "parameters": [
      {
         "name": "query",
         "type": "string",
         "description": "Search query
         for the contacts list."
      },
      {
         "name": "relationship",
         "type": "string",
         "description": "Relationship
         with the person in the contacts
         list to filter by."
      },
      ...
   ],
   "response_schemas": {
      "success": [
         {
            "contact_id": 1,
            "first_name": "string",
            "last_name": "string",
            "email": "user@example.com",
            ...
         }
      ],
      "failure": {
         "message": "string"
      }
   }
}
```

Listing 2: An example of ambiguous API documentation from AppWorld dataset. Problematic parameters are highlighted in red.

Listing 2 shows a case from AppWorld. The SearchContacts API accepts both query and relationship, but the documentation is unclear: it is unspecified whether query accepts only names or also phone numbers/emails, and whether relationship terms should be passed in query or separately in relationship. Such ambiguities complicate reliable inference of parameter-level connections.

B Domain Pairs Excluded by Rule-based Filtering

Figure 6 shows the domain pairs excluded during rule-based filtering. For NESTful, which spans 15 heterogeneous domains, we remove implausible combinations such as *Coronavirus→Spotify*. Two experts independently reviewed the domain list and agreed on mutually incompatible domain pairs, ensuring that only source–target pairs with potential parameter-level connections remain in the dataset.

C Fine-tuning Setup for API Graph Construction on In-N-Out

We fine-tuned models on the In-N-Out dataset to improve API graph construction accuracy. For NESTful, training was run for 30 epochs, while AppWorld was trained for 10 epochs. Checkpoints were selected based on validation performance to avoid overfitting. Listing 3 summarizes the core hyperparameters used in our LoRA-based

Source \ Target	SkyScRapper	Tripadvisor	NewsApi	Reddit	LocalBusinessData	AlphaVantage	CipherCircuit	Real-Time	WeatherAPI	Coronavirus	SpotifyScraper	Instagram	SEC	CountryCodes	Goodreads
SkyScrapper	o	o	o	o	o	o	o	x	o	o	x	o	x	o	o
Tripadvisor	o	o	o	o	o	o	o	x	o	o	x	o	x	o	o
NewsApi	o	o	o	o	o	o	x	o	o	o	o	o	o	o	o
Reddit	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o
LocalBusinessData	x	x	o	o	o	o	o	o	o	o	x	o	o	o	x
AlphaVantage	x	x	o	o	x	o	o	x	x	x	x	x	o	x	x
CipherCircuit	x	x	x	x	x	x	o	x	x	x	x	x	x	x	x
Real-Time	x	x	o	o	x	o	o	o	x	x	x	x	x	x	o
WeatherAPI	x	x	o	o	x	x	x	x	o	o	x	x	x	o	x
Coronavirus	x	x	o	o	x	x	o	x	x	o	x	x	x	o	x
SpotifyScraper	x	x	o	o	x	x	o	o	x	x	o	o	x	o	o
Instagram	o	o	o	o	o	x	o	o	o	o	o	o	o	o	o
SEC	x	x	o	o	o	o	o	x	x	x	x	x	o	x	x
CountryCodes	o	x	o	x	o	x	x	o	x	o	x	x	x	o	x
Goodreads	x	x	o	o	x	x	x	x	x	x	x	x	x	x	o

Figure 6: Irrelevant Domain Pairs for Domain Filtering list. Domain pairs are filtered by *Source→Target* flow, retaining only cases where a source API's output can serve as a target API's input.

fine-tuning setup. For Llama-3.2-3B, Llama-3.1-8B, Qwen2.5-7B, and Qwen2.5-32B, the best validation epochs were 20/18/26/28 for NESTful, and 3/1/2/3 for AppWorld, respectively.

```
gradient_accumulation_steps: 16
max_grad_norm: 1.0
optimizer:
  type: Adam8bit
  params:
    learning_rate: 1.0e-5
    betas: [0.9, 0.995]
    weight_decay: False
LoRA Finetuning (PEFT):
  lora_alpha: 16
  lora_dropout: 0.1
  lora_r: 64
```

Listing 3: Fine-tuning configuration details.

D Examples of Filtered-out Pairs

Shown below are pairs removed by Semantic Filtering: two NESTful examples followed by two AppWorld examples.

Instagram::Following().profile_pic_url → LocalBusinessData::BusinessReviews().language
similarity: 0.32

SEC::CashFlows().filing_filingDate → Tripadvisor::SearchHotels().sort
similarity: 0.48

Phone::ShowAlarms().repeat_days → Venmo::ShowAccount().access_token
similarity: 0.35

Todoist::ShowTask().section_id → Gmail::MarkThreadStarred().email_thread_id
similarity: 0.43

Shown below are pairs removed by Context Aware Filtering: two NESTful examples followed by two AppWorld examples.

Instagram::SearchReels().user_id → SpotifyScraper::GetTrackMetadata().trackId
relevance_score: 0.0

Reddit::PostsByUsername().post_subreddit → SpotifyScraper::GetTrackMetadata().trackId
relevance_score: 0.0

Amazon:SearchProducts().variations_color → Splitwise::UpdatePaymentComment().comment
relevance_score: 0.2

FileSystem::ShowAccount().last_logged_in → venmo::DownloadBankTransferReceipt(). access_token
relevance_score: 0.0

E Prompt Templates

The following are all prompt templates used in our study:

- Documentation Refinement (§3.1): Figure 7.
- Context-Aware Filtering (§3.2): Figure 9.
- API graph Construction (§4.1): Figure 8.
- Generating API call sequences for AppWorld Test Set (§4.2): We adopt the original ReAct prompt from (Trivedi et al., 2024).
- Tool Retrieval with API Graphs (§4.2): Figure 10.
- Structured API Subset Selection for Multi-Tool Query Generation (§4.3): Figure 11.

Documentation Refinement

You are a helpful assistant for a developer who is trying to understand the usage of an API.

The developer will provide you with the API name, description, input parameters, and output parameters. Your job is to generate a description for specific output parameters of the API.

The description must be concise, and help the developer understand the information that the output parameter provides. Do not generate information that is not present in the API documentation.

API: {api_documentation}

I need a description for the output parameter {output_parameter_name}.

Table 7: Prompt template for Documentation Refinement task (§3.1).

API Graph Constrcution

You are a helpful assistant that classifies whether a connection (edge) between two API parameters is valid and natural.

You will be given:
– A source API (with an output parameter)
– A target API (with an input parameter)

[Criteria]
1. Data Compatibility: Can the source output consistently be used as the target input (i.e., do they refer to the same kind of entity/information)?
– Always: classify as “compatible”.
– Only in some cases: classify as “conditional”.
– Not at all: classify as “incompatible”.
2. Naturalness: Is it natural, based on common user behavior, to pass the source output into the target input?
– Natural: “natural”.
– Not natural: “unnatural”.

[Edge label]
– compatible & natural ⇒ “strong-edge”
– conditional & natural ⇒ “weak-edge”
– otherwise ⇒ “non-edge”

[Output format (single JSON object)]
{
“source_api”: “name_of_source_api”,
“target_api”: “name_of_target_api”,
“source_param”: “name_of_source_output_param”,
“target_param”: “name_of_target_input_param”,
“edge_type”: “strong-edge” | “weak-edge” | “non-edge”
}

Now, classify the following edge:
Source API Information: {source_api_documentation}
Target API Information: {target_api_documentation}
Edge to classify: {source_parameter_name} → {target_parameter_name}

Table 8: Prompt template for API Graph Construction task (§4.1)

Context-Aware Filtering

You are an assistant who helps developers understand the relevance between two APIs.

The developer will provide you with the documentation of the APIs. One is the source API, and the other is the target API. Your job is to determine if the specified output parameter of the source API can give full information to the specified input parameter of the target API.

Then you need to generate a relevance score between 0 and 1, which indicates the probability of the output parameter being relevant to the input parameter.

There are some rules you should consider:
1. Prioritize parameter descriptions: Do not rely solely on parameter names to infer their meaning. Always refer to the parameter description, as names can be misleading.
2. Complete information matching: A source parameter can be linked to multiple target parameters if it contains all necessary information. However, partial matches are not allowed.
3. Type compatibility is flexible: Parameter types do not have to be identical as long as the value meaning is preserved.

Source API : {source_api_documentation}
Target API : {target_api_documentation}
Source Parameter: {source_parameter_information}
Target Parameter:{target_parameter_information}

Table 9: Prompt template for Context-Aware Filtering task (§3.2)

Tool Retrieval

You are a helpful assistant that retrieves relevant APIs to obtain a specific input parameter for a given API.
- You are given a task instruction, the documentation of a primary API, and a list of additional available APIs.
- You are also given the name and description of a specific input parameter for the primary API.
- Your goal is to determine how to obtain the value for this input parameter, using the available APIs if necessary.

Your output should be in the following JSON format:
{
"observation": "A brief description of how you can obtain the parameter.",
"prerequisite_api": "The name of the API you need to call to obtain this parameter, if applicable."
}

Instructions:
- If the parameter can be directly inferred from the task instruction or assumed as a default, explain this in 'observation' and leave 'prerequisite_api' as an empty string.
- If the parameter requires calling another API to obtain, describe this in 'observation' and specify the required API in 'prerequisite_api'.

Now, please analyze the task instruction, the primary API's documentation, the target input parameter, and the list of additional available APIs.

Task Instruction: {task_instruction}
Primary API Documentation: {api_documentation}
Target Input Parameter: {target_parameter_name}
Additional APIs available: {top-5 relevant_api_documentation}

Table 10: Prompt template for Tool Retrieval task (§4.2)

Structured API Subset Selection for Multi-Tool Query Generation

You are a helpful assistant that selects small sets of APIs that match a strict dependency pattern.

[Pattern]
- APIs: 1,2,3,4,5
- Edges (must be satisfied internally): {EDGE_LINES: 1→2, 2→3, 3→4, 4→5}

[Task]
From api_list_block, identify up to 5 distinct valid groups of 5 APIs that satisfy the above pattern. In this pattern, an edge is valid only if an output parameter of one API provides the full information required by the corresponding input parameter of another API. Do not output the pattern or any explanation — only the group mappings.

[Connections]
You are provided with connection_list, which are known helpful relationships.
However, it is not mandatory that all required edges appear there. You may also check input-output compatibility.

[Output Format]
Output only in the following format (for each valid group):
APIs: 1: <API 1>, 2: <API 2>, 3: <API 3>, 4: <API 4>, 5: <API 5>

[Rules]
- Replace <API num> with the exact API names from api_list_block.
- Keep the numbering 1..5 exactly as shown.
- Separate multiple groups with a line containing only '—'.
- Do not output placeholders like "app1.api1"; use real API names only.
- Do not add extra commentary, JSON, or text outside the specified block.
- Do not repeat identical (1..5) groups.

api_list_block: {api_list_block}
connection_list: {connection_list}

Table 11: Prompt template for Structured API Subset Selection for Multi-Tool Query Generation. In the no-graph setting, experiments were conducted with the gray text omitted. (§4.3)