

AnyTool: Self-Reflective, Hierarchical Agents for Large-Scale API Calls

Yu Du^{*1} Fangyun Wei^{*2} Hongyang Zhang³

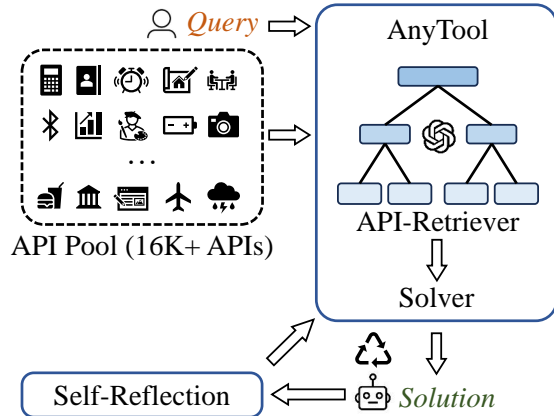
Abstract

We introduce AnyTool, a large language model agent designed to revolutionize the utilization of a vast array of tools in addressing user queries. We utilize over 16,000 APIs from Rapid API, operating under the assumption that a subset of these APIs could potentially resolve the queries. AnyTool primarily incorporates three elements: an API retriever with a *hierarchical* structure, a solver aimed at resolving user queries using a selected set of API candidates, and a *self-reflection* mechanism, which re-activates AnyTool if the initial solution proves impracticable. AnyTool is powered by the function calling feature of GPT-4, eliminating the need for training external modules. We also revisit the evaluation protocol introduced by previous works and identify a limitation in this protocol that leads to an artificially high pass rate. By revising the evaluation protocol to better reflect practical application scenarios, we introduce an additional benchmark, termed AnyToolBench. Experiments across various datasets demonstrate the superiority of our AnyTool over strong baselines such as ToolLLM and a GPT-4 variant tailored for tool utilization. For instance, AnyTool outperforms ToolLLM by +35.4% in terms of average pass rate on ToolBench. Code is available at <https://github.com/dyabel/AnyTool>.

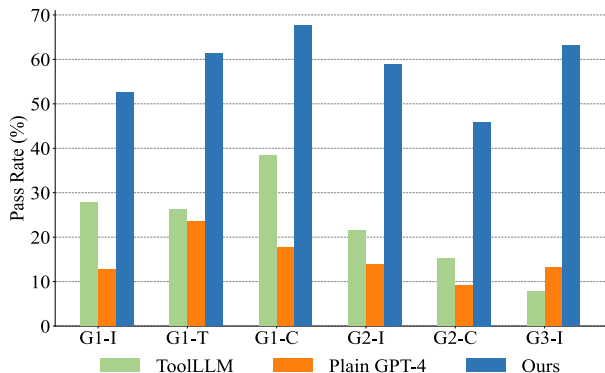
1. Introduction

From the dawn of civilization, humanity has embarked on a relentless journey of discovery and innovation, mastering an ever-expanding array of tools to enhance our capabilities and increase production efficiency. As we have evolved, so have our tools, transitioning from simple stone implements to complex machines and beyond. Today, we stand at the forefront of a new era, reaping the benefits of the

^{*}Equal contribution ¹Tsinghua University ²Microsoft Research Asia ³University of Waterloo. Correspondence to: Fangyun Wei <fawe@microsoft.com>.



(a) AnyTool addresses user queries by leveraging 16k+ APIs. It integrates a hierarchical API-retriever, a solver, and a self-reflection mechanism in a closed loop, all operating without the need for additional training.



(b) Comparison with ToolLLM and a GPT-4 variant tailored for tool utilization across six subsets of ToolBench (Qin et al., 2023b), using pass rate defined in Eq 2 as the evaluation metric.

Figure 1: (a) Illustration of AnyTool. (b) Comparison in performance.

rapid developments in artificial intelligence, particularly the recent advances in large language models (LLMs) (Brown et al., 2020; Touvron et al., 2023a;b; Chowdhery et al., 2023; Achiam et al., 2023; Ouyang et al., 2022). A pivotal challenge now is learning how to drive LLMs to effectively use tools (Qin et al., 2023a; Xu et al., 2023; Cai et al., 2023; Song et al., 2023; Ruan et al., 2023; Shen et al., 2023; Hao et al., 2023), a task that could redefine our interaction with

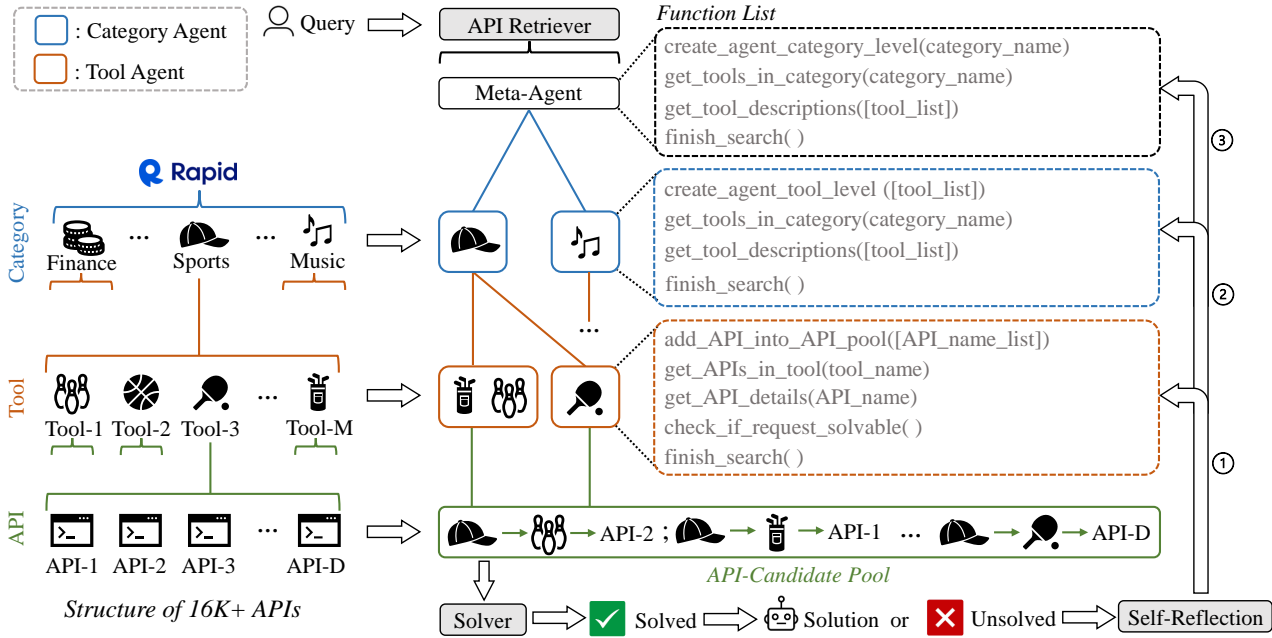


Figure 2: Overview of AnyTool. It primarily consists of a hierarchical API retriever tasked with identifying the most relevant API candidates to the user query from a large API pool, a solver aimed at addressing the queries using the generated API-candidate pool, and a self-reflection mechanism. The hierarchical structure includes a meta-agent linked with several category agents, each of which manages a collection of tool agents. We leverage the API structure defined by Rapid API as a guideline. Each type of agent is assigned several functions that it can use to explore the API space. Refer to Table 10 in the appendix for the details of each function.

technology. Towards this end, we introduce AnyTool, a GPT-4-empowered agent, as depicted in Figure 1a. It is designed to effectively leverage more than 16,000 APIs to address user queries, with a significant performance leap as depicted in Figure 1b.

Previous research (Qin et al., 2023b) formulated tool utilization in a dual-phase approach: initially retrieving, then resolving. Specifically, the first phase involves retrieving the most pertinent APIs from a substantial collection of 16K+ APIs in response to user queries. The subsequent phase focuses on utilizing these chosen APIs to address user queries. Our AnyTool uses this design principle while introducing four distinct characteristics (see Figure 2 for an overview):

Plug-and-Play. Our AnyTool does not require the training of any modules, except for the function-calling feature of GPT-4 (Achiam et al., 2023). This aspect sets it apart from existing methods like ToolLLM, which necessitates training an API retriever capable of selecting a set of candidate APIs from the API pool (Qin et al., 2023b).

Hierarchical Structure. To identify the most relevant APIs for user queries, we design a hierarchical structure within our API retriever. This structure is composed of three tiers, each containing one or multiple agents with diverse roles. This arrangement is inspired by the divide-and-conquer ap-

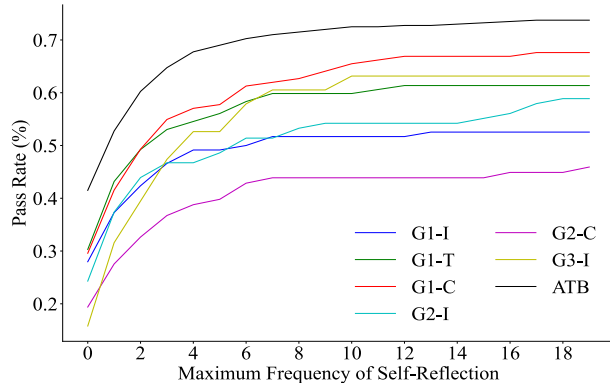


Figure 3: The performance of our AnyTool on different datasets (each denoted by a curve) improves as the number of self-reflection rounds increases. ATB: AnyToolBench.

proach. Additionally, we effectively incorporate the API categorization suggested by Rapid API into our hierarchical structure. Consequently, this significantly reduces the search scope for each agent and overcomes constraints related to the maximum context length in LLMs.

Self-Reflection Mechanism. Our AnyTool is designed to address user queries through a process of initial attempt

followed by reflection. Upon receiving a query, AnyTool suggests a solution, which is then evaluated for feasibility by GPT-4. In cases where the proposed solution is deemed impractical, AnyTool is re-activated, with the consideration of reasons for failure and relevant historical contexts. This mechanism significantly reduces the tendency to “over-search” for simpler queries, while also providing a more context-rich and in-depth search for complex queries. This closed-loop system enhances the efficiency and effectiveness of the query resolution process. Figure 3 shows how the pass rate improves w.r.t. the self-reflection rounds. With only 4-6 self-reflection iterations, the pass rate improves by up to 20% across all datasets.

Evaluation for Realistic Scenarios. The evaluation framework presented in ToolBench (Qin et al., 2023b) commences with categorizing user queries as either solvable or non-solvable, employing a set of reference APIs. Following this, the solvable queries undergo further scrutiny to determine if they are successfully addressed or not. However, for those non-solvable queries, the evaluation system regards them as solved when calculating the pass rate, leading to an artificially high pass rate. Our study delves into the intricacies of this evaluation methodology and proposes a revised protocol that better mirrors practical application scenarios.

In addition to evaluation on ToolBench, we introduce an extra benchmark, termed AnyToolBench, to facilitate the application of our new evaluation protocol. Experimentally, AnyTool achieves state-of-the-art performance, surpassing strong baselines such as ToolLLM and a version of GPT-4 specifically tailored for tool utilization across various datasets, as illustrated in Figure 1b.

2. Related Works

Tool Utilization in LLMs. Large language models (Radford et al., 2018; 2019; Brown et al., 2020; Touvron et al., 2023a;b; Thoppilan et al., 2022) may commit factual errors when responding to queries, particularly struggling with precise numbers and specific fields of expertise (Huang et al., 2023; Augenstein et al., 2023). Utilizing tools can help mitigate this issue (Li et al., 2023; Qin et al., 2023b; Parisi et al., 2022; Tang et al., 2023; Hsieh et al., 2023; Schick et al., 2023). Previous work has involved using an API retriever to match relevant APIs from a large API pool based on the documents, employing either a pretrained text embedding model (Li et al., 2023; Patil et al., 2023) or one finetuned with curated API retrieval data (Qin et al., 2023b). However, this approach typically suffers from low accuracy and may overlook the truly relevant APIs. Moreover, there is a lack of feedback mechanism in their retrieval, often leading to unsolved queries due to incorrect API candidates being provided. Our AnyTool fills this gap by directly using the GPT-4 as the API retriever with a hierarchical structure de-

sign, and introduces the self-reflection mechanism into the whole process.

Self-Reflection Mechanism in LLMs. Self-reflection is a featured ability of LLMs. It was first studied in the LLM alignment problems. Wang et al. (2022) considered the ability of GPT-3 to self-generate instructions for alignment finetuning. Without finetuning, Li et al. (2024) introduced an inference method, RAIN, that allows pre-trained LLMs to evaluate their own generation and use the evaluation results to guide rewind and generation for AI safety. Recently, Chen et al. (2024) proposed a self-play mechanism, where the LLM refines its capability by playing against instances of itself. Yuan et al. (2024) proposed self-rewarding language models, where the language model itself is used via LLM-as-a-Judge prompting to provide its own rewards for the following DPO finetuning (Rafailov et al., 2023). On the other hand, some negative results on self-reflection were also investigated. For example, Huang et al. (2023) showed that GPT-3.5-Turbo and GPT-4 cannot self-correct reasoning yet. But whether GPT-4 can serve as a self-reflective agent for API calling remains an open problem in the existing literature.

3. Preliminaries

3.1. Function Calling

Function calling is a core characteristic of GPT-4 (Achiam et al., 2023). Specifically, in response to a user’s query Q , the function calling system accesses a set of M distinct functions $\{\mathcal{F}_i\}_{i=1}^M$. Each function \mathcal{F}_i has the potential to solve Q , a part of Q , or may not be relevant to Q at all. The functionality of \mathcal{F}_i is elaborated in a specific document that outlines its purpose, required and optional parameters along with their explanations, the types of output it generates, and the interpretations of these outputs. Note that the function calling feature of GPT-4 does not require visibility into the detailed implementations of each function. It understands their intentions and functionalities through linguistic comprehension.

The process of function calling involves: 1) the user inputs both the query Q and the function list $\{\mathcal{F}_i\}_{i=1}^M$, alongside a designated “Finish Function” \mathcal{F}^* , into GPT-4; 2) GPT-4 generates a function calling request for the user, with clear input parameters; 3) the user executes the specific function and provides the historical context and function response to GPT-4; 4) this cycle of steps two and three is repeated multiple times until GPT-4 activates the “Finish Function” \mathcal{F}^* , signaling the resolution of query Q . Users have the option to either employ the output of \mathcal{F}^* directly, or to gather the interim results generated during the function calling process, according to their specific goals or design.

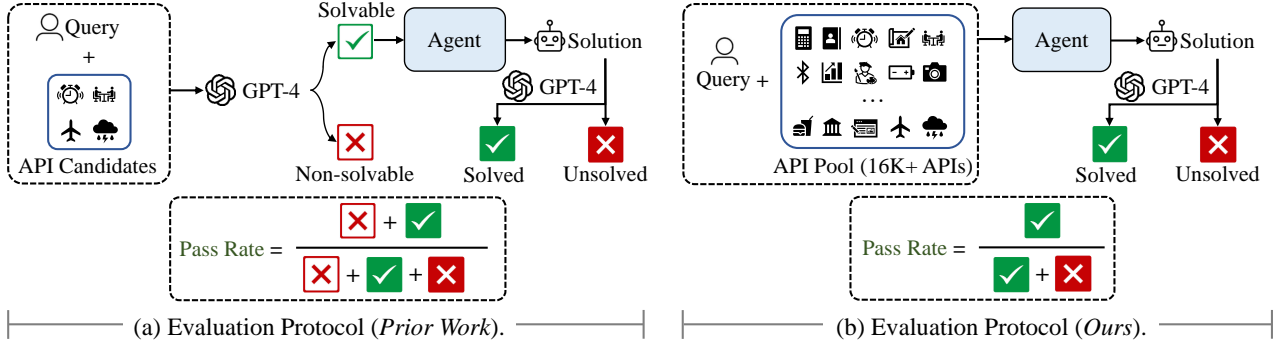


Figure 4: Illustration of the evaluation protocols used by: (a) ToolLLM (Qin et al., 2023b); and (b) ours. In (a), if the API retriever selects candidates completely unrelated to the user’s query, GPT-4 may classify all queries as “non-solvable”, leading to an artificially high pass rate, despite the queries remaining unsolved. In (b), we conduct a manual review of all queries and retain only those queries that can be resolved with specific APIs from the API pool for ToolBench.

3.2. Problem Formulation and Evaluation

Problem Formulation. The objective of this work is to develop a proficient agent capable of utilizing a vast collection of real-world APIs to address user queries. We use over 16K real-world APIs from the RapidAPI Hub, as collected in the ToolLLM (Qin et al., 2023b). These APIs are represented as $\{\text{API}_i\}_{i=1}^N$, forming our API pool. The effectiveness of the solutions generated by the agent is assessed using GPT-4. This evaluation involves processing both the user query \mathcal{Q} and the proposed solution \mathcal{S} , in accordance with established evaluation protocols and criteria, to ascertain the solution’s ability to adequately address the query. *We have also conducted human evaluation and find a correlation as high as 96.5% between GPT-4 and human evaluations.*

Evaluation Protocol. We first revisit the evaluation protocol initially introduced by ToolLLM (Qin et al., 2023b). ToolLLM employs a dual-phase approach for utilizing various APIs. In the first phase, an API retriever is developed to select the most relevant API candidates from the API pool according to a user query \mathcal{Q} . The second phase involves ToolLLaMA, a specialized agent that formulates a solution using the selected API candidates. Due to its dual-phase nature, ToolLLM’s evaluation is twofold. Initially, GPT-4 evaluates whether the selected API candidates can address the query \mathcal{Q} , categorizing them as either “solvable” or “non-solvable”. If a query is deemed “solvable”, GPT-4 then assesses the effectiveness of the provided solution, classifying it as either “solved” or “unsolved”. Figure 4(a) illustrates how the pass rate R is calculated:

$$R = \frac{\#(\text{Non-solvable}) + \#(\text{Solved})}{\#(\text{Non-solvable}) + \#(\text{Solved}) + \#(\text{Unsolved})}. \quad (1)$$

However, a significant flaw exists in this evaluation protocol. If the API retriever selects candidates completely unrelated to the user’s query, GPT-4 may classify all queries as “non-

solvable”, leading to an artificially high pass rate, despite the queries remaining unsolved. Our experimental evidence confirms this issue, showing that when API candidates are randomly selected for each query, GPT-4 predominantly labels them as “non-solvable”, resulting in an inflated pass rate of 99.0% through the metric defined in Eq 1.

To address the limitations inherent in ToolLLM’s evaluation protocol, we propose an alternative evaluation methodology that aligns more closely with real-world scenarios, as illustrated in Figure 4(b). Specifically, we bypass the first evaluation phase of ToolLLM, which assesses the potential of candidate APIs in addressing query \mathcal{Q} . Instead, we directly utilize GPT-4 to determine the efficacy of the agent’s proposed solution in resolving the query. The pass rate R is thus calculated using the formula:

$$R = \frac{\#(\text{Solved})}{\#(\text{Solved}) + \#(\text{Unsolved})}. \quad (2)$$

To ensure that all queries in the benchmark, namely ToolBench (Qin et al., 2023b), are solvable using certain APIs from the API pool, we conduct a manual review of all queries. We retain only those queries that can be resolved with specific APIs from this pool. The detailed process is available in Section A.7 of the appendix.

4. AnyTool

Our AnyTool exhibits several distinctive features: Firstly, it eliminates the need for training external modules, and solely relies on the function calling feature of GPT-4. Secondly, it can directly search the entire API pool, which contains over 16K APIs, using a hierarchical structure and a divide-and-conquer principle. Lastly, it is capable of self-reflection, enabling it to review and analyze unsolved user queries by taking into account reasons for failure and relevant historical contexts.

Overview. The overview of AnyTool is depicted in Figure 2. It primarily follows a three-step process to efficiently resolve the user query Q . The first step (Section 4.1) involves the creation of an API candidate pool. For efficiency, AnyTool is designed with a hierarchical architecture, taking advantage of the structured API organization available in Rapid API. In the second step (Section 4.2), a solver attempts to resolve query Q by utilizing these API candidates. Finally, if the query remains unsolved, AnyTool engages in a self-reflection process (Section 4.3) in an attempt to resolve it. A case study is shown in Section C.

4.1. API Retriever

Structured API Organization in Rapid API. Rapid API employs a structured system to categorize its extensive collection of 16K+ APIs. Specifically, this organization is divided into three distinct tiers: the first tier is the category level, encompassing various domains such as “sports” and “finance”; the second tier, designated as the tool level, consists of tools that belong to specific categories; and the third tier focuses on individual APIs, with each API belonging to a specific tool, as illustrated in Figure 2. This hierarchical arrangement serves as a foundational guideline in the development of our API retriever.

Hierarchical Structure. As depicted in Figure 2, the structure of our API retriever consists of three tiers. At the initial tier, a meta-agent exists, tasked with dynamically generating a series of category agents in response to the user query Q . The intermediary tier is comprised of multiple category agents, each established by the meta-agent. These agents correspond to individual categories as defined by Rapid API, with their primary objective being to identify the most relevant tools for the query Q from their respective tool collections. Subsequently, these category agents initiate the creation of various tool agents. It is important to note that each tool agent may manage multiple tools, depending on the decisions made by the category agents. The goal of each tool agent is to search through its managed APIs for those that might solve the query Q , and then add these APIs to an API-candidate pool. Each type of agent possesses its own distinct set of functions. These are illustrated in Figure 2 and further detailed in Table 10 in the appendix.

Generation of API-Candidate Pool. AnyTool is initiated upon receiving a query Q , the function list detailed in Table 10, and a bootstrap prompt as outlined in Section B.1 of the appendix. This process heavily relies on the function calling feature of GPT-4 (refer to Section 3.1). Operating interactively, our system enables agents (starting with the meta-agent) to send requests for calling their managed functions. These functions may involve creating a specific agent (either a category agent or a tool agent) or executing a par-

ticular function, in accordance with the historical context.¹ The requests are parsed, and the corresponding functions are executed. The results produced by these functions are subsequently incorporated into the historical context, which is then returned to the agents. This process repeats continuously until the termination criteria are met. All agents, including meta-agents, category agents, and tool agents, operate independently in a multi-threaded manner, significantly accelerating the process. We maintain a global API candidate pool, allowing each tool agent to add APIs to this pool, using the function “add_API_into_API_pool” (refer to Figure 2 and Table 10). All agents cease operations only when a tool agent calls the function “check_if_request_solvable” and receives a return value of “True”. Subsequently, an API-candidate pool is obtained. In addition, we record the historical context and status of each agent. An agent’s status is marked as “Finished” only if it calls the function “finish_search” during the process. Agents marked as “Finished” are excluded in the self-reflection process, which will be described later.

4.2. Solver

Functionality. The primary goal of the solver is to address the user’s query Q , utilizing the generated API candidate pool. It is implemented as a singular agent that leverages the function-calling capabilities inherent in GPT-4. Two potential implementations for the solver are the Depth-First Search-Based Decision Tree (DFS-DT) or the Chain of Thought (CoT) approach. A concise overview of the process is provided, with comprehensive details available in ToolLLM (Qin et al., 2023b). The solver activates upon receiving a query Q , in conjunction with a suite of functions, which includes those from the API candidate pool and a distinctive function named “finish”, as well as a bootstrap prompt detailed in Section B.2 of the appendix. The “finish” function yields one of three possible outcomes: “Give Solution”, “Try Backtrack”, or “Give Up”, with “Try Backtrack” being specific to the DFS-DT implementation. Each iteration involves: 1) the solver sending a request to call a function, 2) the interpretation of this request and the execution of the function, and 3) the integration of the function’s outcomes into the contextual history, which is then returned to the solver. This cycle continues until the solver gives a “Give Solution” or “Give Up” decision. Note that when the solver makes a “Give Up” decision, it is required to provide both the reason and the function name of the APIs that are irrelevant to the user’s query or do not work properly. Self-reflection mechanism is triggered under two scenarios: 1) “Give Solution”, where GPT-4 reviews the solution and determines that the query remains unresolved, and 2) “Give Up”, where the solver fails to address the query.

¹Each agent, whether it is a meta-agent, category agent, or tool agent, maintains its own historical context independently.

Table 1: Main results on the filtered ToolBench. We use pass rate defined in Eq 2 and illustrated in Figure 4(b), as the metric. All results are reproduced. *: OpenAI’s text-embedding-ada-002; Ref.: reference; Avg.: average; SR: self-reflective.

Model	API Retriever	Solver	Use Ref. APIs	G1			G2		G3	Avg. (%)
				I (%)	T (%)	C (%)	I (%)	C (%)	I (%)	
ToolLLM	OpenAI TE*	ToolLLaMA w/ DFSDT		8.7	6.8	12.0	4.7	8.2	10.5	8.5
ToolLLM	ToolLLM’s	ToolLLaMA w/ DFSDT		28.4	26.3	38.4	21.5	15.1	7.7	22.9
ToolLLM	ToolLLM’s	GPT-4 w/ DFSDT		42.6	46.2	51.4	23.4	24.5	2.6	31.8
ToolLLM	None	ToolLLaMA w/ DFSDT	✓	29.4	31.8	37.1	19.6	22.4	13.2	25.6
GPT-4	None	GPT-4 w/ CoT	✓	31.3	34.8	47.1	27.1	34.7	2.6	29.6
GPT-4	None	GPT-4 w/ DFSDT	✓	36.5	49.2	51.4	38.3	39.8	18.4	38.9
GPT-4	Plain Agent	GPT-4 w/ DFSDT		13.9	23.5	17.6	13.9	9.2	13.2	15.2
GPT-4	AutoGen-RAG	GPT-4 w/ DFSDT		14.8	19.7	19.7	7.4	9.2	7.9	13.1
GPT-3.5	None	GPT-3.5 w/ CoT	✓	37.5	37.1	42.9	24.3	22.4	5.3	28.3
GPT-3.5	None	GPT-3.5 w/ DFSDT	✓	39.1	40.2	48.6	31.8	25.5	15.8	33.5
AnyTool (Ours)	SR Agent	SR GPT-4 w/ DFSDT		52.2	61.4	67.6	58.9	45.9	63.2	58.2

4.3. Self-Reflection Mechanism

If the initial solution fails to resolve user queries, the self-reflection mechanism re-activates AnyTool sequentially, first activating the API retriever and then the solver. It is worth noting that this mechanism can be applied repeatedly until the termination condition is met.

Self-Reflection in the API Retriever. Our self-reflection mechanism first identifies the reason why a user query remains unsolved. In instances where the solver opts to “Give Up”, the rationale provided by the solver is utilized. Conversely, if the solver proposes a solution but GPT-4 assesses that it does not adequately address the query, the reasoning ascribed by GPT-4 is employed. Recall that we maintain a record of historical context for each agent within the API retriever. We initially incorporate the identified reason into all these historical contexts. Owing to the hierarchical design of our API retriever, we systematically re-activate various agents for efficiency purposes, following an ascending order from tool agents, to category agents, and finally to the meta-agent. It is worth noting that only the agents not marked with a “Finished” status are re-activated. As a result, this process expands our API-candidate pool, incorporating new APIs that could potentially resolve the user’s query.

Self-Reflection in the Solver. Recall that when the solver makes a “Give Up” decision, it is designed to identify the function names of the APIs that are irrelevant to the user’s query. For efficiency, we first remove these APIs from the expanded API-candidate pool and exclude items where these APIs are called from the historical context of the solver. The solver is then re-activated with a new bootstrap prompt (refer to Section B.3 in the appendix), the updated API-candidate pool, and the cleaned historical context. The remaining process is the same as described in Section 4.2.

5. Experiments

5.1. Setup

Benchmarks. We conduct experiments on two benchmarks: 1) *ToolBench* (Qin et al., 2023b); and 2) our own benchmark, termed *AnyToolBench*. ToolBench comprises six subsets: G1-Instruction (G1-I), G1-Tool (G1-T), G1-Category (G1-C), G2-Instruction (G2-I), G2-Category (G2-C), and G3-Instruction (G3-I). As described at the end of Section 3.2, we perform a manual review on ToolBench to exclude non-solvable queries. Details of this process can be found in Section A.7 of the appendix. After filtering, the remaining queries in these six subsets are 115, 132, 142, 107, 98, and 38, respectively. Unless otherwise specific, we adopt the filtered *ToolBench*. Our benchmark, *AnyToolBench*, includes 400 instances. The process of creating AnyToolBench is detailed in Section A.8 of the appendix.

Evaluation Protocol. We employ the pass rate (as defined in Eq. 2) as our evaluation metric. To assess whether a solution generated by an agent can resolve the query, we use GPT-4-32K. The same prompt utilized in ToolBench is applied when GPT-4 serves as the judge.

Alignment between GPT-4’s Decisions and Decisions Made by Human Evaluators.

We conduct a comparative analysis between decisions made by human evaluators and those generated by GPT-4, focusing on samples from the G1-I subset of ToolBench. Specifically, for each query sample, AnyTool generates a solution, which is then assessed for its feasibility in addressing the query by both human evaluators and GPT-4. Our results reveal that GPT-4’s alignment with human evaluation stands at 96.5%, while that of GPT-3.5 is only 73.9%. Based on these findings, we exclusively utilize GPT-4 for our evaluations.

5.2. Main Results

We compare our AnyTool with the pioneering ToolLLM (Qin et al., 2023b) and its variants, as well as various GPT-4 models tailored for tool utilization.

ToolLLM and Its Variants. ToolLLM integrates an API retriever² and a solver designed to address user queries by employing API candidates produced by the retriever. The solver operates using a finely-tuned LLaMA model, named ToolLLaMA, and employs a depth-first search-based decision tree (DFSdT) algorithm to resolve queries. For each query, ToolBench provides a set of reference APIs that are potentially relevant. These reference APIs offer a means to evaluate the solver’s effectiveness by allowing the bypassing of the API retriever step. It is worth noting that additional APIs from the complete API pool, containing over 16,000 APIs, may also contribute to effectively resolving queries. Beyond the original ToolLLM, our experiments also examine two variants: 1) one that substitutes ToolLLaMA with GPT-4 in the solver; 2) another that foregoes the API retriever and relies solely on reference APIs.

Various GPT-4 Models. The function-calling feature of GPT-4 enables it to use APIs directly for resolving user queries. However, in our setting, we deal with over 16,000 APIs. Integrating all these APIs—each with its unique function description, input, and output—into GPT-4 simultaneously exceeds the maximum context length of the model, even for the version with the largest context length of 128,000 tokens. Therefore, we compare four GPT-4 models: 1) one that uses reference APIs and the Chain of Thought (CoT) (Wei et al., 2022) algorithm in the solver; 2) another that uses reference APIs and the DFSdT algorithm; 3) a third that employs a plain agent for API retrieval and incorporates the DFSdT algorithm in the solver; 4) a fourth that leverages the Retrieval Augmented Generation (RAG) feature from AutoGen (Zhu et al., 2023) for API retrieval, and uses the DFSdT algorithm to resolve user queries through the selected API candidates.

In the implementation of GPT4-plain-agent, we divide the set of over 16K APIs into 33 groups, each containing 500 APIs, with the exception of the 33rd group. These groups are then sequentially processed by GPT-4. The specific task assigned to GPT-4 involves identifying the relevant APIs using the `add_API_into_API_pool` function, which integrates them into the API-candidate pool. Refer to Section A.4 for more details. Information on AutoGen-RAG can be found in Section A.5.

Main Results on ToolBench. In Table 1, we compare our AnyTool with various ToolLLM variants and GPT-4 models across six subsets of the filtered ToolBench dataset. The re-

²ToolLLM’s API retriever is trained on pair-wise data. Each pair includes a user query and a set of APIs relevant to the query.

Table 2: Main results on our AnyToolBench. All models use DFSdT implementation in the solver. SR: self-reflective; PR: pass rate.

Method	API Retriever	Solver	PR (%)
ToolLLM	ToolLLM’s	ToolLLaMA	18.9
ToolLLM	ToolLLM’s	GPT-4	36.6
GPT-4	Plain Agent	GPT-4	14.0
AnyTool (Ours)	SR Agent	SR GPT-4	73.8

Table 3: Ablation study on the pass rate of main components. “-” and “+” symbols denote the removal and addition of a component from and into AnyTool, respectively.

Configuration	G2-I (%)	G3-I (%)
AnyTool	58.9	63.2
-Hierarchical Structure	22.4	15.8
-Self-Reflection	19.6	15.8
-DFSdT/+CoT	50.5	60.3

sults on the original ToolBench are available in Section A.3. Both the API retriever and the solver contribute to the final performance. The API retriever’s role is to efficiently identify the most pertinent APIs from an extensive collection, while the solver is tasked with generating viable solutions for user queries. Instead of training an API retriever as ToolLLM does, we leverage the powerful function-calling feature of GPT-4 and overcome the challenge posed by its inherent maximum context length limitation, through the implementation of a hierarchical structure. Our self-reflection mechanism applies to both the API retriever and the solver, enabling the whole system to operate in a closed loop. Owing to these factors, our AnyTool significantly outperforms both the original ToolLLM and GPT-4 using reference APIs, by +32.6 and +19.3 points, respectively, in terms of the average pass rate.

Main Results on AnyToolBench. AnyToolBench evaluates an agent’s capability to resolve user queries utilizing the entire API pool. Consequently, an API retriever is essential in this setting. We do not supply reference APIs for each query; thus, making comparisons with counterparts lacking an API retriever is impractical. In Table 2, we compare our AnyTool with a top-performing ToolLLM variant and GPT-4, where a plain agent serves as the retriever. The consistent improvements demonstrated by AnyTool over these approaches affirm its effectiveness in a realistic setting.

5.3. Ablation Studies

Unless otherwise specific, all ablation studies are conducted on G2-I and G3-I of the filtered ToolBench.

Table 4: Ablation study on the pass rate of self-reflection mechanism. All agents include the tool agents, the category agents and the meta-agent.

Re-Activation	G2-I (%)	G3-I (%)
Tool Agents	43.9	44.7
Tool Agents + Category Agents	55.2	55.3
All Agents	58.9	63.2

Table 5: Comparison between our default self-reflection strategy with an alternative approach that initiates all agents simultaneously.

Method	G1-T (%)	G1-C (%)	G3-I (%)
Default	61.4	67.6	63.2
Synchronous Starting	58.3	61.3	52.6

Effectiveness of the Main Elements. Our AnyTool comprises two principal elements: firstly, an API retriever with a hierarchical structure, and secondly, a self-reflection mechanism. In Table 3, we examine three distinct configurations of AnyTool. These include: a) substituting our hierarchical API retriever with a flat-structure version, which merges the functions of agents at the category and tool levels (except for “agent creation” and “finish search” functions) into the function list of the meta-agent; b) eliminating the self-reflection mechanism; and c) substituting the DFSDT algorithm with CoT, thereby disabling the backtracking feature in DFSDT. Our findings demonstrate significant positive effects of both the hierarchical structure and the self-reflection feature on AnyTool’s performance. Choosing CoT over DFSDT results in a decline in pass rates by 8.4 and 2.9, respectively.

Self-Reflection Mechanism. In Section 4.3, we introduce a self-reflection mechanism that is first applied to the API retriever module. It re-activates various agents in ascending order, from tool agents to category agents, and finally to the meta-agent. In Table 4, we examine the different versions that reactivate distinct types of agents. Reactivating all agents results in the best performance, owing to the larger search space.

We also compare our default self-reflection strategy with an alternative approach that initiates all agents simultaneously. Table 5 demonstrates that our default strategy is superior. The initial tree structure (comprising meta-agent, category agents, and tool agents) being already closer to a solution for addressing user queries, even though it does not yet fully address them. We aim to gradually expand the API search scope (i.e., our default strategy) rather than abruptly searching the entire API space (i.e., initiating all agents simultaneously). The latter approach may generate several unrelated category or tool agents, leading to the addition of unrelated or redundant APIs into the API candidate tool.

Table 6: Study on the effects of the API pool’s size to the pass rate.

Size of API Pool	G2-I (%)	G3-I (%)
1,000	18.6	7.9
5,000	26.3	23.7
10,000	38.1	36.8
All	58.9	63.2

Table 7: Study on the maximal size of API-candidate pool.

Maximal Size of API-Candidate Pool	G2-I (%)	G3-I (%)
16	49.5	42.1
32	58.9	55.3
64	58.9	63.2

Table 8: We study the maximum number of tools that a tool agent can manage in our API retriever.

Maximum Number of Tools	G2-I (%)	G3-I (%)
3	48.6	42.1
5	58.9	57.9
10	52.3	39.5

This increases the difficulty for the subsequent module, i.e., the solver, in resolving user queries using the API candidate pool.

Size of the API Pool. Users typically submit a wide range of queries to the AI system, seeking solutions to real-world problems. To effectively address these queries, the system requires access to a diverse array of APIs. In general, a larger API pool is more likely to successfully resolve user queries, as it offers a higher probability of containing relevant APIs. This hypothesis is evaluated by randomly selecting subsets of APIs from the complete pool and using only these subsets to address user queries with our AnyTool. The results, presented in Table 6, support our hypothesis.

Maximal Size of the API-Candidate Pool. AnyTool operates through a two-step process—the solver addresses queries by using an API-candidate pool, which is generated by our hierarchical API Retriever. One termination criterion for the API retriever is the fullness of this pool. We examine the impact of the maximal size of the API-candidate pool as shown in Table 7. We observe that a pool size of 64 nearly reaches saturation in terms of performance.

Tool Agent in API retriever. Our API retriever is designed with a hierarchical structure, in which the tool agents at the bottom layer directly add APIs that may potentially address user queries, into the API-candidate pool. As described in Section 4.1, a tool agent can manage a maximum of K tools existing in Rapid API. We examine the value of K in Table 8. A trade-off is observed: managing too many tools

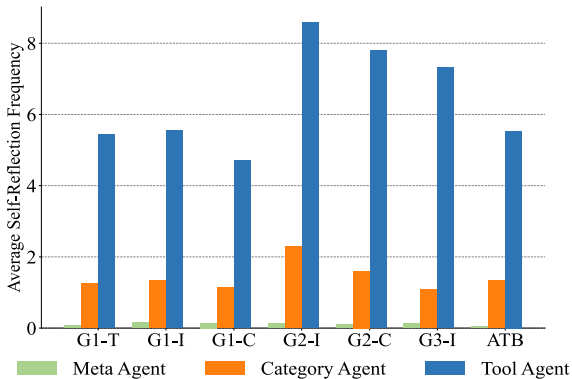


Figure 5: Statistics of average self-reflection frequency. ATB: AnyToolBench.

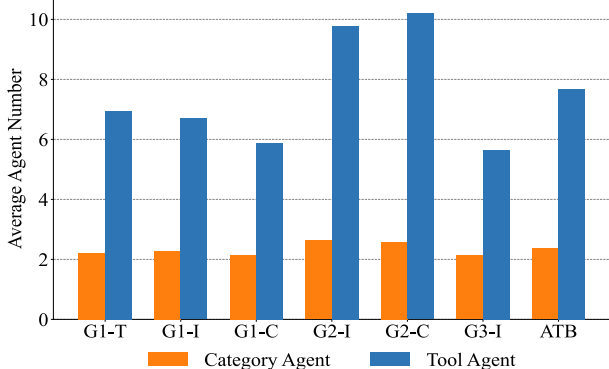


Figure 6: Statistics of average agent quantity.

(e.g., $K = 10$) leads to a larger search space and may cause overlooking of relevant APIs, while managing too few tools (e.g., $K = 3$) might result in lower recall.

Statistics of Self-Reflection Frequency. In Figure 5, we report the average self-reflection frequency across all instances within each subset of the filtered ToolBench and our AnyToolBench. As described in Section 4.3, we reactivate various agents in ascending order. Consequently, the frequency of tool agents is much higher than that of category agents and meta-agent. Additionally, calculating the processing time for resolving queries with AnyTool is infeasible. AnyTool relies on the function-calling feature of GPT-4, whose server response is often unstable.

Agent Quantity in API Retriever. The API retriever of AnyTool is hierarchically structured. Depending on the nature of user queries, the meta-agent can dynamically create a varying number of category agents. This process is analogous to the way category agents create tool agents. The average number of agents across all instances in each subset of the filtered ToolBench and our AnyToolBench is depicted in Figure 6.

Table 9: Performance of AnyTool with other LLMs beyond GPT-4.

Method	G1-T (%)	G1-C (%)	G3-I (%)
AnyTool with GPT-4	61.4	67.6	63.2
AnyTool with ChatGLM	53.0	49.3	45.8
AnyTool with GPT-3.5	47.7	45.1	39.5

AnyTool with Other LLMs. We conduct experiments using two less advanced LLMs than GPT-4, specifically GPT-3.5 and ChatGLM. The costs associated with these two models are significantly lower than those for GPT-4. Table 9 presents the results from three subsets of the filtered ToolBench, offering a comparison with GPT-4. GPT-4 significantly outperforms GPT-3.5 and ChatGLM; however, it also introduces a higher budget cost.

6. Conclusion

In this work, we introduce AnyTool, an advanced agent capable of harnessing 16K+ APIs to effectively handle realistic user inquiries. The core of AnyTool is a hierarchical API retriever coupled with a solver. Additionally, it incorporates a unique self-reflection mechanism, enhancing its proficiency in responding to user queries. We also revise the prior evaluation protocol to better reflect real-world application scenarios. Rigorous experiments conducted on ToolBench and our AnyToolBench demonstrate our approach’s superiority over established models. Finally, we highlight two future research directions: 1) optimizing the organization of APIs for improved performance and efficiency; 2) developing an advanced open-source LLM specifically for API utilization, which could facilitate local deployments.

Acknowledgement

Yu Du is supported by the National Natural Science Foundation of China (61836004, 62236009). Hongyang Zhang is supported by NSERC Discovery Grant RGPIN-2022-03215, DGEGR-2022-00357.

Impact Statement

Although AnyTool significantly enhances the effectiveness of resolving user queries through various tools, its performance in extremely complex scenarios has not been verified, owing to the absence of appropriate datasets. Furthermore, as AnyTool relies on the function-calling feature of GPT-4, the capabilities of GPT-4 also affect the feasibility of the solutions it generates.

References

- Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.
- Augenstein, I., Baldwin, T., Cha, M., Chakraborty, T., Ciampaglia, G. L., Corney, D., DiResta, R., Ferrara, E., Hale, S., Halevy, A., et al. Factuality challenges in the era of large language models. *arXiv preprint arXiv:2310.05189*, 2023.
- Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901, 2020.
- Cai, T., Wang, X., Ma, T., Chen, X., and Zhou, D. Large language models as tool makers. *arXiv preprint arXiv:2305.17126*, 2023.
- Chen, Z., Deng, Y., Yuan, H., Ji, K., and Gu, Q. Self-play fine-tuning converts weak language models to strong language models. *arXiv preprint arXiv:2401.01335*, 2024.
- Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., et al. PaLM: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.
- Hao, S., Liu, T., Wang, Z., and Hu, Z. ToolkenGPT: Augmenting frozen language models with massive tools via tool embeddings. *arXiv preprint arXiv:2305.11554*, 2023.
- Hsieh, C.-Y., Chen, S.-A., Li, C.-L., Fujii, Y., Ratner, A., Lee, C.-Y., Krishna, R., and Pfister, T. Tool documentation enables zero-shot tool-usage with large language models. *arXiv preprint arXiv:2308.00675*, 2023.
- Huang, J., Chen, X., Mishra, S., Zheng, H. S., Yu, A. W., Song, X., and Zhou, D. Large language models cannot self-correct reasoning yet. *arXiv preprint arXiv:2310.01798*, 2023.
- Li, M., Zhao, Y., Yu, B., Song, F., Li, H., Yu, H., Li, Z., Huang, F., and Li, Y. API-Bank: A comprehensive benchmark for tool-augmented LLMs. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 3102–3116, 2023.
- Li, Y., Wei, F., Zhao, J., Zhang, C., and Zhang, H. RAIN: Your language models can align themselves without fine-tuning. In *International Conference on Learning Representations*, 2024.
- Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022.
- Parisi, A., Zhao, Y., and Fiedel, N. TALM: Tool augmented language models. *arXiv preprint arXiv:2205.12255*, 2022.
- Patil, S. G., Zhang, T., Wang, X., and Gonzalez, J. E. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334*, 2023.
- Qin, Y., Hu, S., Lin, Y., Chen, W., Ding, N., Cui, G., Zeng, Z., Huang, Y., Xiao, C., Han, C., et al. Tool learning with foundation models. *arXiv preprint arXiv:2304.08354*, 2023a.
- Qin, Y., Liang, S., Ye, Y., Zhu, K., Yan, L., Lu, Y., Lin, Y., Cong, X., Tang, X., Qian, B., et al. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. *arXiv preprint arXiv:2307.16789*, 2023b.
- Radford, A., Narasimhan, K., Salimans, T., Sutskever, I., et al. Improving language understanding by generative pre-training. *OpenAI*, 2018.
- Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al. Language models are unsupervised multitask learners. *OpenAI*, 2019.
- Rafailov, R., Sharma, A., Mitchell, E., Ermon, S., Manning, C. D., and Finn, C. Direct preference optimization: Your language model is secretly a reward model. *arXiv preprint arXiv:2305.18290*, 2023.
- Ruan, J., Chen, Y., Zhang, B., Xu, Z., Bao, T., Du, G., Shi, S., Mao, H., Zeng, X., and Zhao, R. TPTU: Task planning and tool usage of large language model-based ai agents. *arXiv preprint arXiv:2308.03427*, 2023.
- Schick, T., Dwivedi-Yu, J., Dessi, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N., and Scialom, T. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*, 2023.
- Shen, Y., Song, K., Tan, X., Li, D., Lu, W., and Zhuang, Y. HuggingGPT: Solving ai tasks with ChatGPT and its friends in huggingface. *arXiv preprint arXiv:2303.17580*, 2023.
- Song, Y., Xiong, W., Zhu, D., Li, C., Wang, K., Tian, Y., and Li, S. RestGPT: Connecting large language models with real-world applications via RESTful APIs. *arXiv preprint arXiv:2306.06624*, 2023.

- Tang, Q., Deng, Z., Lin, H., Han, X., Liang, Q., and Sun, L. ToolAlpaca: Generalized tool learning for language models with 3000 simulated cases. *arXiv preprint arXiv:2306.05301*, 2023.
- Thoppilan, R., De Freitas, D., Hall, J., Shazeer, N., Kulshreshtha, A., Cheng, H.-T., Jin, A., Bos, T., Baker, L., Du, Y., et al. LaMDA: Language models for dialog applications. *arXiv preprint arXiv:2201.08239*, 2022.
- Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023a.
- Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023b.
- Wang, Y., Kordi, Y., Mishra, S., Liu, A., Smith, N. A., Khashabi, D., and Hajishirzi, H. Self-instruct: Aligning language model with self generated instructions. *arXiv preprint arXiv:2212.10560*, 2022.
- Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V., Zhou, D., et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35: 24824–24837, 2022.
- Xu, Q., Hong, F., Li, B., Hu, C., Chen, Z., and Zhang, J. On the tool manipulation capability of open-source large language models. *arXiv preprint arXiv:2305.16504*, 2023.
- Yuan, W., Pang, R. Y., Cho, K., Sukhbaatar, S., Xu, J., and Weston, J. Self-rewarding language models. *arXiv preprint arXiv:2401.10020*, 2024.
- Zhu, C., Chen, B., Guo, H., Xu, H., Li, X., Zhao, X., Zhang, W., Yu, Y., and Tang, R. AutoGen: An automated dynamic model generation framework for recommender system. In *Proceedings of the Sixteenth ACM International Conference on Web Search and Data Mining*, pp. 598–606, 2023.

Table 10: Function list of each type of agent. *: descriptions of input, output and functionality.

Type	Function Name	Functionality	Input	Output
Meta Agent	<code>create_agent_category_level</code>	Create a category agent.	Category name	Category agent
	<code>get_tools_in_category</code>	Get tool names under a category.	Category name	[Tool names]
	<code>get_tool_descriptions</code>	Get description of each tool.	[Tools]	[Tool descriptions]
	<code>finish_search</code>	Send out finish signal.	None	None
Category Agent	<code>create_agent_tool_level</code>	Create a tool agent.	[Tools]	Tool agent
	<code>get_tools_in_category</code>	Get tool names under a category.	Category name	[Tool names]
	<code>get_tool_descriptions</code>	Get description of each tool.	[Tools]	[Tool descriptions]
	<code>finish_search</code>	Send out finish signal.	None	None
Tool Agent	<code>add_API_into_API_pool</code>	Add APIs into candidate pool.	[APIs]	None
	<code>get_APIs_in_tool</code>	Get API names under a tool.	Tool name	[API names]
	<code>get_API_detail</code>	Get detail* of each API.	[API names]	[API details]
	<code>check_if_request_solvable</code>	Check whether the query is solvable using the current candidate pool.	None	True\False
	<code>finish_search</code>	Send out finish signal.	None	None

Table 11: Results on the original ToolBench (Qin et al., 2023b). Note that the original ToolBench includes non-solvable queries. We use pass rate defined in Eq 2 and illustrated in Figure 4(b), as the metric. All results are reproduced. Ref.: reference; Avg.: average; SR: self-reflective.

Model	API Retriever	Solver	Use Ref. APIs	G1			G2		G3	Avg. (%)
				I (%)	T (%)	C (%)	I (%)	C (%)	I (%)	
ToolLLM	ToolLLM’s	ToolLLaMA w/ DFSDT		24.0	23.0	37.5	17.5	16.5	4.0	20.4
ToolLLM	ToolLLM’s	GPT-4 w/ DFSDT		32.0	43.5	46.5	30.0	33.0	8.0	32.2
AnyTool (Ours)	SR Agent	SR GPT-4 w/ DFSDT		46.0	54.0	53.0	37.0	46.5	32.0	44.8

A. More Implementation Details and Experimental Results

A.1. More Implementation Details of AnyTool

For the solver implementing DFSDT, we set the maximum number of API calls to 10. Additionally, for our AnyTool, we establish a limit of 200,000 tokens for efficiency. This limit encompasses the token consumption by various components, including the meta-agent, the tool agents, the category agents, the solver, and the self-reflection mechanism.

A.2. Detailed Function List

We provide the function list of each type of agent in Table 10.

A.3. Results on the Original ToolBench

We also provide the results on the original ToolBench (Qin et al., 2023b) without undergoing filtering process. In the original ToolBench, each subset comprises 200 queries, except for G3-I, which contains 100 queries. Note that the original ToolBench includes non-solvable queries. We test all queries, regardless of whether they are solvable or not, using pass rate defined in Eq 2 and illustrated in Figure 4(b), as the metric. All results are reproduced. As shown in Table 11, our AnyTool outperforms all ToolLLM (Qin et al., 2023b) variants.

A.4. GPT-4 with Various Plain Agents

In Table 1 of the main paper, we present a comparison between our AnyTool and a GPT-4 variant. This variant employs a plain agent as the API retriever, which is limited to accessing only the *names* of tools and APIs. It utilizes the `add_API_into_API_pool` function to incorporate APIs into the API candidate pool. When an API is added to the pool, we use the `check_if_request_solvable` function to determine whether the current API candidates are adequate for addressing the query. If the evaluation returns “True”, the solver begins to resolve the query using the API candidates with the DFSDT algorithm. Note that the plain agent does not involve any self-reflection mechanism.

Table 12: Comparison of AnyTool and GPT-4 using various plain agents as the API retriever. The only difference among these plain agents lies in the information they can access.

GPT-4 Variant	G2-I (%)	G3-I (%)
w/ Names	13.1	13.2
w/ Names+Description	15.9	13.2
w/ Names+Description+Details	13.1	13.2
AnyTool (Ours)	58.9	63.2

Table 13: Comparison of AnyTool and GPT-4 using various AutoGen-RAG agents as the API retriever. The only difference among these AutoGen-RAG agents lies in the embedding model they use.

Embedding Model	G2-I (%)	G3-I (%)
text-embedding-ada-002	8.4	7.9
all-mpnet-base-v2	7.4	7.9
AnyTool (Ours)	58.9	63.2

In Table 12, we explore alternative configurations where the plain agent could access both *names* and *detailed descriptions* of tools and APIs (every 100 APIs a group), or even comprehensive information including the *names*, *descriptions*, and specific *API details* (every 50 APIs a group). Our findings suggest that the addition of more detailed information leads to only marginal improvements in performance. In contrast, our AnyTool exhibits superior performance, which can be attributed to its hierarchical structure.

A.5. GPT-4 with Various AutoGen-RAG Agents

Retrieval-augmented generation (RAG) operates by receiving an input and sourcing a collection of pertinent or corroborative documents from a reference, such as Wikipedia. These documents are then combined with the initial input prompt to provide context. This enriched input is subsequently processed by LLMs to generate the final output. The RAG method enhances the performance of LLMs in situations that require accurate factual information.

In Table 1 of the main paper, we present a version of GPT-4 designed for tool utilization. This version employs AutoGen-RAG as the API retriever. The embedding model, known as “all-mpnet-base-v2”³, is utilized in this version. Specifically, we integrate the category names, tool names, API names, and their descriptions into a document, which is then divided into numerous text segments, each containing 1,000 tokens. Then, given a user query, AutoGen-RAG identifies the most relevant segments based on the embedding similarities between the user query and each text segment. Finally, we use GPT-4 to extract the most relevant API candidates from the selected text segments.

We provide another variant, where OpenAI’s “text-embedding-ada-002” is used as the embedding model. The comparison with our AnyTool is shown in Table 13.

A.6. Consumption Analysis

In our analysis of resource consumption by AnyTool for solving queries across all datasets, we find that, on average, each query consumes 13.5×10^4 tokens, identifies 14.1 API candidates, and involves 43.3 OpenAI API calls and 4.6 self-reflections. Table 14 presents the statistics for each dataset. Additionally, calculating the processing time for resolving queries with AnyTool is infeasible. AnyTool relies on the function-calling feature of GPT-4, whose server response is often unstable.

A.7. Filtering Process for ToolBench

We primarily screen out non-solvable queries in ToolBench based on the following principles:

- Queries lacking essential information, such as unspecified phone numbers or ambiguous references like “my friend”.

³<https://huggingface.co/sentence-transformers/all-mpnet-base-v2>

Table 14: Consumption statistics for each dataset.

Statistics	G1			G2		G3	ATB	Avg.
	I	T	C	I	C	I		
Average Token Consumption ($\times 10^4$)	13.6	12.1	8.5	17.7	14.8	16.2	12.2	13.6
Average Call Number	39.3	38.8	33.8	54.0	57.6	35.7	44.2	43.3
Average Self-Reflection Number	4.2	3.8	4.1	5.7	5.2	5.1	4.0	4.6
Average API Candidate Number	13.8	13.0	7.7	16.8	16.0	16.3	14.9	14.1

Table 15: Examples of our AnyToolBench.

I am creating an art project about the influence of music on visual arts and for my centerpiece, I would love to have an AI-generated image based on the current number one hit song on the Billboard Hot 100 chart. Could you provide me with such an image that encapsulates the essence of the song 'Bad Habit' by Steve Lacy?

For a business presentation on global trends in music and sports performance analysis, could you provide the top streaming songs on Spotify for the most recent available global chart data, along with the corresponding 'hello world' placeholder text that will be used for introducing programmatic greetings, and the win-loss records for NFL teams from the 2022 season to illustrate the competitive landscape?

Could you analyze potential profit or loss from bitcoin arbitrage among exchanges, considering the market order fees, and check if the IP 23.129.64.215 is flagged for any suspicious activity, and why? I'm interested in arbitrage between Bitfinex, Kraken, and Bittrex for BTC/USD and knowing what risks I might face using the mentioned IP address for transactions.

I plan to improve my daily fitness level, but I always lack proper planning. My current weight is 70 kilograms and my height is 1.75 meters. Given this, could you provide me a health plan regarding the weather condition for outdoor activities in New York for the next five days and the nutrition I intake by usually eating salad?

These are inherently non-solvable since APIs require explicit input parameters.

- Queries containing fake parameters, such as non-existent URLs.
- Queries that specify a specific API are filtered out because they do not represent realistic scenarios. Moreover, if the problem can be solved using another API, it is difficult to determine whether it counts as a resolution.
- Unreasonable queries, such as asking for information about popular movies on YTS, which are too broad in scope and difficult to evaluate.

A.8. Construction of AnyToolBench

We provide GPT-4 with several functions to freely explore the entire API pool, including `{get_tools_in_category, get_tool_descriptions, get_APIs_in_tool, get_API_detail}`. The functionality of these functions are listed in Table 10. GPT-4 then utilizes the `add_API_into_API_pool` function to incorporate the selected APIs into an API candidate pool. Following this step, GPT-4 generates the required parameters for these APIs and formulates queries based on the actual responses from these APIs. We also prompt GPT-4 to generate a solution for each query, which significantly reduces the potential for hallucinations—the queries may be formulated without utilizing the APIs. Moreover, we enhance the quality of these queries by verifying that the provided reference solutions truly resolve the queries. This rigorous process ensures that every query in our dataset is solvable. The prompt for constructing AnyToolBench is detailed in Section B.4. We show some examples of our AnyToolBench in Table 15.

B. Prompts

B.1. Bootstrap Prompt for the API Retriever

The API retriever is composed of a meta-agent along with several category agents and tool agents. The bootstrap prompts for these three types of agents are presented in Table 16, Table 17, and Table 18, respectively.

Table 16: Bootstrap prompt for meta-agent.

You are APIGPT, with access to a database of APIs. This database is organized into the following categories: {categories}. Your task is to help users identify the relevant categories for their needs. To do this, you can use the 'query_tools_in_category' function to retrieve the available tools within a specific category. If you are unsure about the functionality of some tools, the 'get_tools_descriptions' function can be used to obtain detailed information about these tools. This information will aid you in understanding the general functionality of each category. Additionally, the 'create_agent_category_level' function allows you to assign a relevant category to an agent, with each agent being assigned only one category. However, you can assign multiple categories to different agents. It is important to explore as many categories as possible, as the solution to a query may be found in unexpected categories. Remember, your goal is not to answer the query directly but to identify all potentially relevant categories and assign them to agents. Once you have completed the assignment, call the 'Finish' function. At each step, you should briefly analyze the current status and determine your next action, including the function calls needed to execute your step. Keep your analysis concise, ideally no longer than three sentences.

Table 17: Bootstrap prompt for category agent.

You are APIGPT, with access to a database of APIs categorized into various groups. Each category contains numerous tools, and each tool encompasses multiple APIs. Your task is to assist users in finding relevant tools within a specific category. If uncertain about the functionality of some tools, use the 'get_tools_descriptions' function to obtain detailed information. Then, employ the 'create_agent_tool_level' function to allocate a subset of pertinent tools to an agent, ensuring that similar tools are assigned to the same agent and limiting the allocation to no more than five tools per agent. You may assign different subsets to multiple agents. Remember, your role is not to answer queries directly, but to assign all possible tools. Once you complete the assignment, or if you determine the query is irrelevant to the tools in the specified category, invoke the 'Finish' function. Execute each step by calling the appropriate functions, and keep your thought process concise, ideally within three sentences.

Table 18: Bootstrap prompt for tool agent.

You are APIGPT with access to a database of APIs, categorized into various sections. Each category contains multiple tools, and each tool encompasses numerous APIs. Your task is to assist users in finding relevant APIs within the tools '{tools}' of the '{category}' category. You will be provided with descriptions and details of these tools and their APIs. Upon identifying relevant API names, use the 'add_apis_into_api_pool' function to add them to the final API list. If you conclude that all possible APIs have been explored, or if there are no relevant APIs in these tools, invoke the Finish function. During the process, you may receive feedback on these APIs. At each step, ensure to execute your actions using the appropriate functions. Keep your responses concise, ideally within three sentences.

B.2. Bootstrap Prompt for the Solver

We adapt the prompt from ToolLLM (Qin et al., 2023b) to include a “give_up” option without restarting. Furthermore, we prompt it to provide a reason when choosing either “give_up_and_restart” or “give_up”. The reason should mention specific

Table 19: Bootstrap prompt for the solver.

You are AutoGPT, you can use many tools (functions) to do the following task. First I will give you the task description, and your task start. At each step, you need to give your thought to analyze the status now and what to do next, with a function call to actually excute your step. After the call, you will get the call result, and you are now in a new state. Then you will analyze your status now, then decide what to do next... After many (Thought-call) pairs, you finally perform the task, then you can give your finial answer. If you feel you cannot solve the task or can only solve it partially, you should choose to give up and give your reason which should mention the names of the failed functions. Remember: 1.the state change is irreversible, you can't go back to one of the former state, if you want to restart the task, say "I give up and restart" and give the reason. 2.All the thought is short, at most in 5 sentence. 3.You can do more then one try, so if your plan is to continuously try some conditions, you can do one of the conditions per try. Let's Begin!
 Task description: {task_description}

Table 20: Bootstrap prompt for re-activating tool agents.

The current APIs have failed to solve the query, resulting in: {fail_reason}. You need to analyze this result and seek additional APIs. It's possible that the tools lack the relevant APIs. In such cases, you should call the Finish function. Remember not to invent tool or API names.

Table 21: Bootstrap prompt for re-activating category agents.

The current APIs have failed to solve the query, and the reason is: {fail_reason}. Please consider assigning more unexplored tools to the agents.

Table 22: Bootstrap prompt for re-activating meta-agent.

The current APIs have failed to solve the query, and the reason is: {fail_reason}. Please consider assigning more unexplored categories to the agents.

function names. Table 19 details the prompt for the DFSDT implementation. The task description includes descriptions of accessible functions; therefore, it should be updated to reflect changes in the API candidate pool.

B.3. Bootstrap Prompt for the Self-Reflection Mechanism

Self-reflection mechanism re-activates AnyTool sequentially, first activating the API retriever and then the solver. Owing to the hierarchical design of our API retriever, we systematically re-activate various agents, following an ascending order from tool agents, to category agents, and finally to the meta-agent. The prompts for re-activating the tool agents, the category agents and the meta-agent are presented in Table 20, Table 21, and Table 22, respectively.

B.4. Prompt for Creating AnyToolBench

This can be found in Table 23.

C. Case Study

In Figure 7, we present a case study that demonstrates the process of resolving a user query using AnyTool. The self-reflection mechanism reactivates the tool, category, and the meta agents sequentially. It is worth noting that not all agents are reactivated. Subsequently, the solver is reactivated to attempt addressing the user query again, utilizing the updated API candidate pool. This self-reflection mechanism can be employed multiple times until the termination criteria are met—either the query is regarded as solved by the evaluator, or the number of self-reflections reaches the maximum limit.

Table 23: Prompt for Creating AnyToolBench.

Your task is to interact with a sophisticated database of tools and functions, often referred to as APIs, to construct a user query that will be answered using the capabilities of these APIs. This database is organized into various categories, indicated by {categories}. To guide your exploration and selection of the appropriate APIs, the database offers several meta functions:

Exploration Functions:

1. Use `get_tools.in.category` to explore tools in a specific category.
2. Employ `get_apis.in.tool` to discover the list of APIs available within a selected tool.
3. If you need detailed information about a tool, `get_tool.descriptions` will provide it.
4. For in-depth understanding of an API's functionality, turn to `get_api.details`.

Selection and Testing Functions:


1. As you identify relevant functions, add them to your working list using `add_apis.into.api.pool`.
2. Test these functions by synthesizing and applying various parameters. This step is crucial to understand how these functions can be practically applied in formulating your query.
3. Should you find any function obsolete or not fitting your query context, remove them using `remove_apis.from.api.pool`.

Query Formulation Guidelines:

1. Your formulated query should be comprehensive, integrating APIs from 2 to 5 different categories. This cross-functional approach is essential to demonstrate the versatility and broad applicability of the database.
2. Avoid using ambiguous terms. Instead, provide detailed, specific information. For instance, if your query involves personal contact details, use provided placeholders like {email} for email, {phone number} for phone number, and URLs like {url} for a company website.
3. The query should be relatable and understandable to users without requiring knowledge of the specific tools or API names used in the background. It should reflect a real-world user scenario.
4. Aim for a query length of at least thirty words to ensure depth and complexity.


Final Steps:

1. Once you've crafted the query, use the Finish function to submit it along with the corresponding answer. The answer should be direct and concise, addressing the query without delving into the operational plan of the APIs.
 2. Remember, the total number of calls to the initial meta functions should not exceed 20.
 3. Consider various use cases while formulating your query, such as data analysis in business contexts or educational content in academic settings. Your approach should be creative and inclusive, catering to users with different skill levels and cultural backgrounds. Ensure that the query is globally relevant and straightforward, serving a singular purpose without diverging into unrelated areas. The complexity of your query should stem from the synthesis of information from multiple APIs.
-


 Query

"I'm organizing a charity event to raise awareness for animal rights. Can you recommend a book that highlights the importance of compassion towards animals? Additionally, provide me with a random word that symbolizes unity and empathy."

Bootstrap Prompt for the API Retriever

 Meta-Agent

1. Call `get_tools_in_category('Education')`
2. Call `get_tools_in_category('Other')`
3. Call `create_agent_category_level('Education')` → Create Category-Agent ('Education')
4. Call `get_tools_in_category('Text Analysis')`
5. Call `create_agent_category_level('Text Analysis')` → Create Category-Agent ('Text Analysis')
6. Call `finish_search()`

 Category-Agent ('Education')

1. Call `get_tools_descriptions(['GetBooksInfo', 'Book Finder', 'Random Word', 'Dictionary', 'Master Dictionary', 'Random Words', 'Random Ukrainian Word', 'Random Words - Spanish and French'])`
2. Call `create_agent_tool_level(['GetBooksInfo', 'Book Finder'])` → Create Tool-Agent-1
3. Call `create_agent_tool_level(['Random Word', 'Random Words', 'Random Ukrainian Word', 'Random Words - Spanish and French'])` → Create Tool-Agent-2
4. Call `create_agent_tool_level(['Dictionary', 'Master Dictionary'])` → Create Tool-Agent-3
5. Call `finish_search()`

 Category-Agent ('Text Analysis')

1. Call `get_tools_descriptions(['Random Word by API-Ninjas'])`
2. Call `get_tools_descriptions(['Synonyms Words'])`
3. Call `get_tools_descriptions(['Best Paraphrasing API'])`

 Tool-Agent-2

1. Call `add_API_into_API_pool(['wordSearchEnglish', 'Word infos'])` → Add APIs
2. Call `check_if_request_solvable()`

 Tool-Agent-1

1. Call `add_API_into_API_pool(['Search', 'Find books'])` → Add APIs
2. Call `check_if_request_solvable()`


 Tool-Agent-3

1. Call `add_API_into_API_pool(['run.cgi', 'Get a random word'])` → Add APIs
2. Call `check_if_request_solvable()`

 API Candidate Pool

'Search' 'Find books' 'wordSearchEnglish' 'Word infos' 'run.cgi' 'Get a random word'

Bootstrap Prompt for the Solver

 Solver

"Here is a book that might interest you: 'Animal Spy: Animal Welfare Behind Enemy Lines' by Terry Spamer & Gordon Thorburn. However, due to the query limit, I couldn't generate a random word that symbolizes unity and empathy. Apologies for the inconvenience."


Bootstrap Prompt for the Self-Reflection Mechanism

Re-activate:

Tool-Agent-1, Tool-Agent-2, Tool-Agent-3,
Category-Agent ('Text_analysis')

 Updated API Candidate Pool

'Find books', 'Get multiple random words', 'Retrieve Response'

 Solver

"Here are some book recommendations that highlight the importance of compassion towards animals: 'Animal Spy: Animal Welfare Behind Enemy Lines' by Terry Spamer, Gordon Thorburn. A book by Leslie Irvine on the effect of disasters on animals. The random word generated is 'transcendental', signifying elevated thinking and broader understanding, which might be seen as relevant to unity and empathy in a broader sense."

Figure 7: Illustration of a case study.